Universidad Autónoma de Madrid
Escuela Politécnica Superior

Trabajo fin de Máster

DESARROLLO DE UN ALGORITMO ANT COLONY OPTIMIZATION PARA TAREAS DE CLUSTERING EN APACHE SPARK

Master en Ingeniería Informática

Autor: Ortiz Martin, Alejandro
Tutor: González Pardo, Antonio

Fecha: FEB 2016

DESARROLLO DE UN ALGORITMO ANT COLONY OPTIMIZATION PARA TAREAS DE CLUSTERING EN APACHE SPARK

AUTOR: Alejandro Ortiz Martin
TUTOR: Antonio González Pardo
David Camacho Fernández

Grupo de la EPS: AIDA
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
FEB 2016

Resumen

Resumen

En los últimos años se ha producido un incremento en la cantidad de datos generada por las redes sociales, logs de software, dispositivos móviles y sensores, entre otros. Dicha cantidad de datos es de tal magnitud que se requieren de nuevos paradigmas de computación para el correcto análisis de la información contenida en ellos.

En este entorno ha surgido el área de Big Data que se usa para hacer referencia a los desafíos y ventajas derivadas de la recolección y procesado de grandes cantidades de datos [1]. De una manera más formal, el Big Data se define como la cantidad de datos que exceden las capacidades de cómputo de un determinado sistema en términos de consumo de memoria y/o tiempo[2].

La computación distribuida permite contar con múltiples ordenadores interconectados entre sí formando clusters, consiguiendo una capacidad conjunta mayor que con un único ordenador más potente. En la actualidad existen varios frameworks para el análisis de Big Data que han atraído el interés tanto de la comunidad científica como de la industria. El primer framework es Apache Hadoop [3], desarrollado por Google y que se basa en el enfoque de MapReduce[4]. Sin embargo, el nuevo framework Apache Spark[5], desarrollado por la universidad de Berkeley, se está haciendo bastante popular.

Hacer un buen uso de estos frameworks requiere adaptar los algoritmos que se quieran usar a las características del sistema sobre el cual se vayan a desplegar, encontrando puntos de paralelización óptimos que aprovechen las fortalezas de dichos frameworks. La correcta adaptación de los algoritmos a la plataforma de Big Data es un aspecto crucial ya que repercutirá en el rendimiento de dicho algoritmo.

Este Trabajo de Fin de Máster se centrará en el estudio y desarrollo algoritmo de clusterización de Ant Colony Optimization (ACOC)[6, 7] sobre la plataforma Apache Spark. Para la correcta validación del sistema desarrollado, se realizarán tareas de clustering sobre varios conjuntos de pruebas sencillos. Una vez que el sistema esté validado y si se dispone de tiempo suficiente, se estudiará el rendimiento del sistema ante un problema de Big Data como pueden ser las tareas de clustering sobre datos de redes sociales como Twitter.

Palabras Clave

Ant Colony Optimization, Ant Colony Optimization for Clustering, Apache Spark, Aprendizaje automático, Computación distribuida

Abstract

In recent years there has been an increase in the amount of data generated by social networks, software logs, mobile devices and sensors, among others. This amount of data is such that require new computing paradigms for proper analysis of the information contained therein.

In this environment it has emerged Big Data. This term is used to refer to the challenges and benefits of collecting and processing large amounts of data[1]. In a more formal way, Big Data is defined as the amount of data that exceed the computing capabilities of a given system in terms of memory consumption and/or time[2].

Distributed computing allows for multiple interconnected computers together to form clusters, achieving a combined capacity greater than a single more powerful computer. At present there are several frameworks for analysis of Big Data that have attracted the interest of both the scientific community and industry. The first one is Apache Hadoop [3], developed by Google and based on the MapReduce approach[4]. However, the new framework Apache Spark[5], developed by the University of Berkeley, is becoming quite popular.

Making good use of these frameworks requires adapting the algorithms that want to use the features of the system on which they will be deployed, finding optimal parallelization points that leverage the strengths of these frameworks. The correct implementation of algorithms for Big Data platform is a crucial aspect as it will affect the performance of the algorithm.

This Final Master Thesis will focus on the study and development of clustering algorithm Ant Colony Optimization (ACO)[6, 7] on the Spark Apache platform. Multiple tests by clustering simple tasks will be performed for proper validation of the developed system. Once the system is validated and if time permits, system performance will be studied with Big Data problems, such as data clustering on social networks data like Twitter.

Key words

Ant Colony Optimization, Ant Colony Optimization for Clustering, Apache Spark, Machine learning, Distributed computing

Índice general

Índice de figuras

Índice de tablas

1 Introducción

En la actualidad, las cantidades de datos que se producen y que es necesario procesar para ser aprovechados es cada vez mayor. Desde hace unos años, el termino *Big Data* se refiere tanto a los datos producidos como a los métodos que logran domar el tamaño y variabilidad de estas cantidades masivas de información en bruto. Nuevos métodos, paradigmas y aproximaciones surgen para tratar de enfrentarse a este problema, que las metodologías tradicionales habían fracasado en abordar con éxito. No solo se busca la capacidad de guardar toda esta información, sino la capacidad de procesarla y aprovecharla, depurando y extrayendo lo más relevante de cada Terabyte.

De entre todos los proyectos que encaran este problema de gestión y procesamiento de información, hay uno que destaca tanto por su carácter open-source como por la calidad de sus resultados. Este proyecto es *Apache Hadoop*. *Hadoop* es un framework de la *Apache Software Foundation* escrito en *Java*, diseñado para distribuir, almacenar y procesar grandes cantidades de datos haciendo uso de clusters de hardware básico. Además, *Hadoop* facilita la integración con otros proyectos de *Apache*, como *Apache Mahout* o *Apache Hive*, que ofrecen implementaciones escalables de muchos algoritmos de aprendizaje automático y análisis de datos.

En los últimos años, otro proyecto de *Apache* derivado de *Hadoop* ha mejorado enormemente las características de su antecesor: *Apache Spark*. *Spark* permite hacer uso de la memoria RAM de las computadoras para alcanzar rendimientos hasta 100 veces mejores que los logrados por *Hadoop*. Además, incluye un módulo llamado *MLlib* que contiene implementaciones de gran cantidad de algoritmos de análisis de datos y aprendizaje automático que aprovechan las características de la arquitectura distribuida de *Spark*.

Todas estos avances han cambiado en muchos aspectos la gestión de datos tradicional. Antes, cuando el tamaño de un problema superaba las capacidades de la máquina, no se contaban con las herramientas de computación distribuida que se han mencionado. Algoritmos especiales tuvieron que ser desarrollados para solucionar problemas que se encontraban más alla de las capacidades de los algoritmos tradicionales. Entre estos nuevos métodos se encuentran los algoritmos bio-inspirados. Este nuevo paradigma computacional, gracias al uso de meta-heurísticas y optimizaciones estocásticas, logra extraer conclusiones y resultados sin tener que explorar por completo todo el espacio de soluciones. Para ello aprovechan un cierto componente aleatorio, junto con procesos iterativos, para simular una evolución y lograr solucionar con éxito problemas demasiado costosos para otros algoritmos. *Ant Colony Optimization* (ACO) es uno de estos algoritmos bio-inspirados, clasificado como inteligencia de enjambre, que se inspira en el

comportamiento de las hormigas al buscar comida.

1.1. Motivación del proyecto

MLlib no incluye ninguna implementación de algoritmos evolutivos, las implementaciones que contiene se centran principalmente en el análisis estadístico de los datos. El objetivo de este Trabajo de Fin de Master es implementar un algoritmo de *Ant Colony Optimization* para tareas de Clustering en *Apache Spark*. A traves de este trabajo se explorará el potencial que un algoritmo evolutivo como ACO puede alcanzar en el entorno distribuido que proporciona *Apache Spark*, aprovechando todo su potencial para distribuir el procesamiento.

1.2. Objetivos del proyecto

Los objetivos del presente Trabajo Fin de Master serán los siguientes:

1. Estudio de los algoritmos de hormigas aplicados a las tareas de clustering.
2. Estudio de los diferentes aspectos del algoritmo ACO que puedan ser distribuibles.
3. Estudio de la plataforma *Apache Spark*, así como de los posibles algoritmos de ACO desarrollados en ella.
4. Desarrollo del algoritmo ACOC para tareas de clustering.
5. Validación del rendimiento del algoritmo diseñado.

2

Estado del arte

Este capítulo tiene como objetivo exponer el estado del arte actual en las áreas en las que se engloba este trabajo. La sección 2.1 proporciona una visión general de la inteligencia computacional, centrándose principalmente en la inteligencia de enjambre y, más concretamente, en el algoritmo Ant Colony Optimization (ACO).

Para complementar este estado del arte, el apartado 2.2 proporciona una aproximación al Big Data, la computación distribuida y el *framework Spark* sobre el cual se desarollará *Ant Colony Optimization* para tareas de Clustering (ACOC) [7].

2.1. Inteligencia computacional

El área denominada Inteligencia Computacional es un subconjunto de la inteligencia artificial que cubre una serie de paradigmas tecnológicos inspirados en diferentes paradigmas que se pueden encontrar en la naturaleza. Podemos distinguir 5 grandes paradigmas: *Artificial Neural Networks* (NN), *Evolutionary Computacion* (EC), *Artificial Immume systems* (AIS), *Fuzzy Systems* (FS) y *Swarm Intelligence* (SW).

Cada uno de estos paradigmas se apoya en técnicas probabilísticas, y se inspira en distintos sistemas biológicos: las redes neuronales artificiales modelan las redes neuronales biológicas, la computación evolutiva se inspira en la evolución natural, los sistemas inmunes artificiales se inspiran en el sistema inmunológico humano, los sistemas difusos se originaron a partir de estudios de la interacción de organismos con su entorno y la inteligencia de enjambre modela el comportamiento social de colonias o enjambres de organismos vivos.

Para este Trabajo de Fin de Master (TFM) se prestará atención especial a la inteligencia de enjambre, área que contiene el algoritmo Ant Colony Optimization que trataremos en detalle en la sección 2.1.2. A pesar de ello, veremos brevemente el resto de áreas con el objetivo de conseguir una visión más ámplia al abordar la inteligencia computacional. Durante el apartado 2.1.1 veremos las distintas areas de la inteligencia computacional: NN, EC, AIS y FS. Durante el apartado 2.1.2 nos centraremos en la inteligencia de enjambre y ACO.

2.1.1. Paradigmas de la inteligencia computacional

Artificial Neural Networks **(NN)**

Las redes neuronales artificiales se inspiran en las redes neuronales biológicas para simular un conjunto de 'neuronas' interconectadas que intercambian datos. El objetivo más habitual de dichas redes es aproximar una función desconocida con un gran número de entradas. Este objetivo se puede lograr gracias a la principal fortaleza de estas redes: su capacidad de aprendizaje.

Estas redes neuronales artificiales simulan el concepto de neurona como una función que de acuerdo a unos estímulos o entradas genera una salida. En la figura la figura 2.1 podemos ver el esquema general de una neurona básica, donde **Z** representan las I entradas y **V** representan sus respectivos pesos o *weights*. Esta salida, o valor neto, suele verse influenciado también por un *bias*, que permite ajustar con mayor exactitud la función de activación. El valor neto de la entrada se obtiene aplicando la fórmula 2.1, y a partir de ese valor neto una función de activación f define si la combinación de las entradas activa la neurona o no. Entre las funciones de activación más usadas se encuentran la sigmoidal o la escalón.

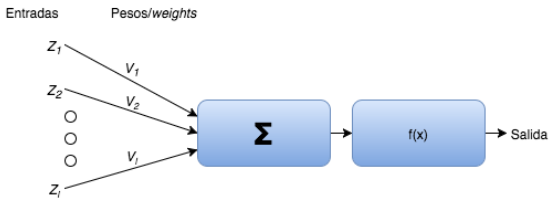


Figura 2.1: Esquema de una neurona básica

$$neto = \sum_{i=0}^{i=I} Z_i \times V_i \tag{2.1}$$

Cuando las salidas de unas neuronas se conectan a las entradas de otras se obtienen las redes neuronales, que dan la posibilidad de simular funciones más complejas. Estas conexiones, tambien denominadas sinápsis, se suelen disponer en forma de capas, como muestra la figura 2.2. Las salidas de las neuronas de una capa sirven como entrada a las neuronas de la siguiente capa, y así sucesivamente. La complejidad de una red se puede aumentar aumentando el número de capas, o aumentando la cantidad de neuronas que cada capa contiene. La memoria de una red neuronal se encuentra en los valores de los pesos **V** que unen las neuronas de que está compuesta.

Los distintos tipos de estas redes se definen principalmente por la arquitectura interna y de los métodos de aprendizaje seguidos. Algunos ejemplos de tipos de redes neuronales son:

- Las redes de Hopfield, de una única capa.
- Los perceptrones multicapa, de múltiples capas y algoritmo de aprendizaje *backpropagation*.
- Las redes de Jordan y Elman, que cuentan con el retraso temporal al ejecutarse.
- Las redes de Kohonen, que son redes autorganizadas con aprendizaje competitivo.

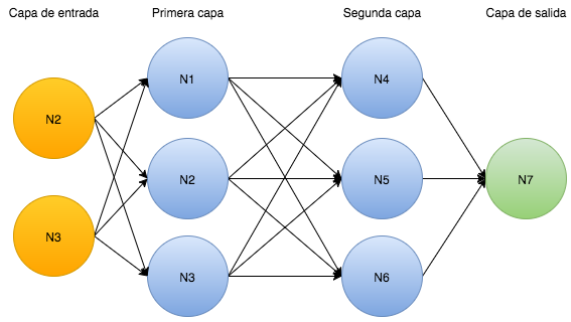


Figura 2.2: Esquema de red neuronal con 2 capas ocultas.

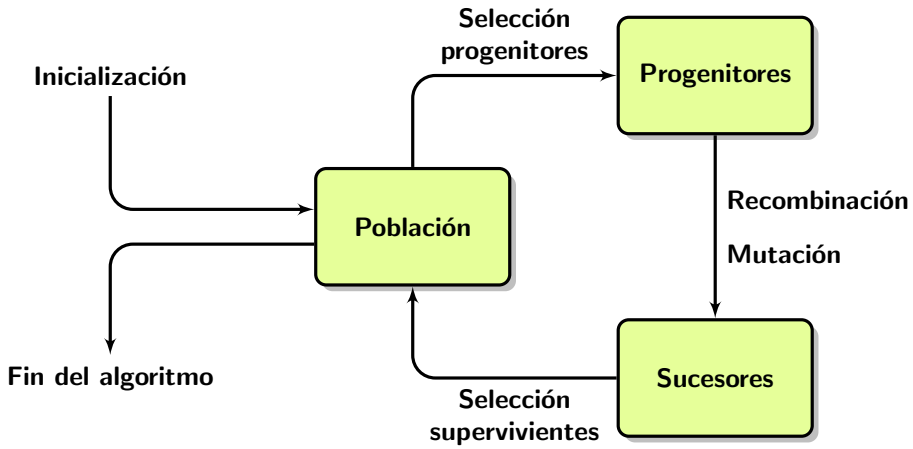


Figura 2.3: Esquema genérico de un algoritmo genético.

Evolutionary Computacion (EC)

La computación evolutiva[8, 9] se comenzó a desarrollar basándose en las ideas de Darwin entre los años 1950 y 1970, y se centra en solucionar problemas de optimización a través de dos métodos usados conjuntamente: metaheurísticas y optimización estocástica. Las metaheurísticas son procedimientos genéricos y abstractos aplicables a un conjunto de problemas que suelen abandonar algún objetivo menor para lograr alcanzar la solución. Normalmente se usan cuando no se conoce un algoritmo que proporcione la solución óptima, o cuando el tiempo necesario para solucionar el problema mediante los algoritmos clásicos es inmanejable. Por otro lado, la optimización estocástica es un tipo de optimización que hace uso de un cierto grado de aleatoriedad, normalmente para evitar tener que explorar todo el espacio de soluciones.

La computación evolutiva suele hacer uso de una población sobre la que se aplica una progresión iterativa, simulando un crecimiento o desarrollo.

Los algoritmos genéticos son la rama más conocida de la computación evolutiva y se inspiran en la teoría de la selección natural que Darwin propuso. La idea general de estos algoritmos es simular una población donde cada individuo representa una solución del problema a tratar. Para ello es necesario tener una representación genética del dominio de soluciones (en forma de cadena de datos, llamado genotipo) y una función de fitness, también llamada función de evaluación, que sea capaz de evaluar cada una de las posibles representaciones. A lo largo de las iteraciones, la evolución de la población se produce al utilizar el material genético de los individuos de una generación para producir los individuos que formarán la siguiente población, por medio de los operadores de reproducción: cruce y mutación (Ver esquema 2.3).

El operador de recombinación o crossover produce uno o dos sucesores a partir de dos

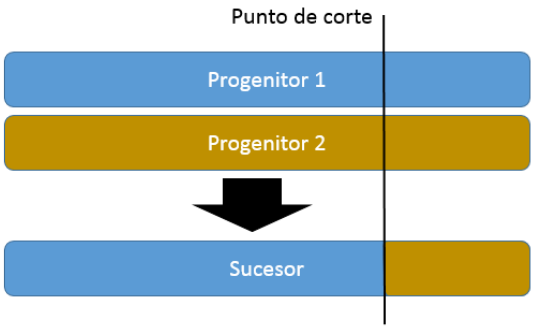


Figura 2.4: Ejemplo de crossover en un punto.

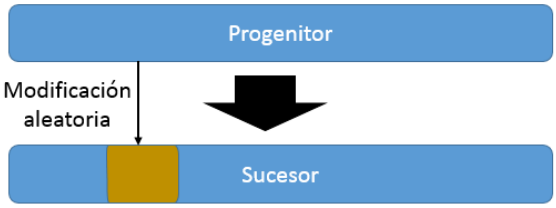


Figura 2.5: Ejemplo de mutación.

progenitores. La versión del crossover más utilizada se denomina crossover en un punto, y consiste en cortar los genotipos de los progenitores por un punto aleatorio y formar los sucesores a partir de la combinación de los trozos resultantes como muestra la figura 2.4.

El operador de mutación modifica un sucesor generado por crossover. Para esta modificación se escoge un gen del sucesor aleatoriamente y se modifica su valor, como se observa en la figura 2.5.

Los operadores de selección también tienen una gran importancia en la definición del algoritmo genético. Se aplican tanto en seleccionar los progenitores como en seleccionar la nueva población a partir de la antigua y de los nuevos individuos. A continuación se listan algunos de los más comunes métodos de selección de individuos:

- **Selección aleatoria**: Cada individuo tiene una probabilidad de ser seleccionado de $\frac{1}{n_s}$, donde n_s es el tamaño de la población. Dado que no se usa la función de fitness, los mejores y los peores individuos tienen la misma posibilidad de ser seleccionados.

- **Selección proporcional**: En este caso, cada individuo tiene una probabilidad de ser seleccionado proporcional a la calidad de su solución, aplicando la fórmula 2.2:

$$\varphi_s(\mathbf{x}_i(t)) = \frac{f(\mathbf{x}_i(t))}{\sum_{l=1}^{n_s} f(\mathbf{x}_l(t))} \tag{2.2}$$

  Donde n_s es el número de invididuos de la población, $f(\mathbf{x}_i)$ es el fitness de $\mathbf{x}_i$ y $\varphi_s(\mathbf{x}_i(t))$ es la probabilidad resultante.

- **Elitismo**: Se seleccionan los mejores individuos. Aplicando únicamente este método suele perderse demasiada diversidad de la población, por lo que se suele aplicar complementando alguno de los métodos anteriores. Por ejemplo, al determinar que individuos formarán la nueva población, se seleccionan los 10 mejores individuos de la población anterior y el resto de la población se selecciona con Selección proporcional.

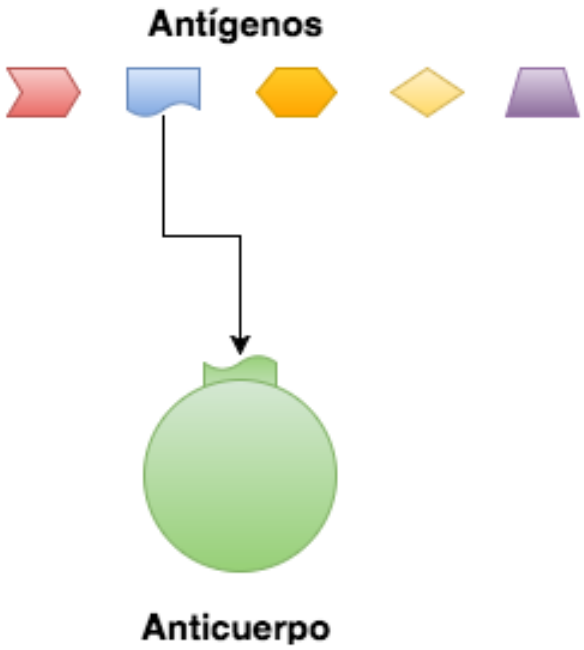


Figura 2.6: Representación gráfica de un anticuerpo y los posibles antígenos a neutralizar.

Los algoritmos genéticos se ejecutan hasta que se satisface un criterio de aceptación. Este criterio puede ser sencillo, como un límite de tiempo o un número de iteraciones fijas. Si se hace uso de estos límites, debe tenerse en cuenta que el algoritmo genético debe tener tiempo suficiente para explorar el espacio de soluciones.

Además de estos criterios límite, suele buscarse alcanzar la convergencia de la población. Se considera que la población ha convergido cuando ya no se producen cambios significativos en la misma. Esto suele detectarse por diversos métodos: cuando no hay mejora en el fitness medio población en varias generaciones, cuando no hay mejora en el mejor fitness de la población en varias genraciones, o cuando se alcanza un fitness objetivo.

Artificial Immume systems (AIS)

Los algoritmos del campo de los sistemas inmunes artificiales se inspiran en el sistema inmunológico humano, y tienen como punto fuerte su capacidad de adaptación, a través de los mecanismos de memoria y aprendizaje extraídos de dicho sistema.

Los dos conceptos clave para entender el funcionamiento del sistema inmune humano son antígeno y anticuerpo. Un antígeno es cualquier sustancia que provoque una reacción del sistema inmune, generalmente la producción de anticuerpos. Un anticuerpo es una molécula que identifica y neutraliza elementos extraños del organismo, en nuestro caso los antígenos. Cada uno de estos anticuerpos identifica y elimina un tipo de antígeno. En la figura 2.6 vemos una representación aproximada, donde el anticuerpo solo neutraliza los antígenos que 'encajan' o identifica. Este procedimiento provoca que si llega un antígeno nuevo que ningún anticuerpo identifica, éste no es eliminado.

El cuerpo humano es capaz de generar alrededor de 10 mil millones de anticuerpos distintos, lo que nos permite neutralizar la mayor parte de los elementos nocivos extraños a nuestro organismo. El objetivo de los sistemas inmunes artificiales es simular esta generación de anticuerpos, buscando uno que logre neutralizar el antígeno adecuado, que se considera la solución.

Para ello, al desarrollar un algoritmo AIS hay 4 decisiones que se deben tomar: Codificación, Medida de similitud, Mutación y Selección.

La codificación será completamente dependiente del problema en cuestión. Tanto antígenos como anticuerpos se codifican del mismo modo, idealmente en forma de cadena y de forma similar a la codificación en el caso de la computación evolutiva. El objetivo será encontrar un anticuerpo con una codificación lo más similar posible al del antígeno a neutralizar.

La medida de similitud sirve entre otras cosas para saber si se ha hallado el anticuerpo buscado. Esta medida es completamente dependiente de la codificación seleccionada, y cumple una función similar a la función de fitness en los algoritmos evolutivos, ya que indica cómo de cerca estamos de tener la solución óptima.

La mutación nos permite explorar el espacio de soluciones a partir de lo ya explorado, modificando anticuerpos en vez de generarlos aleatoriamente por completo. Al igual que los casos anteriores, depende de la codificación y del problema a solucionar, al igual que en los algoritmos evolutivos.

El tipo de selección elegida suele determinar el subtipo de algoritmo AIS que se implementa. Es importante seleccionar el tipo de selección adecuado en función del tipo de problema que se quiera solucionar. Entre los principales tipos de selección se encuentran:

- ***Clonal Selection***: Este tipo de selección se suele aplicar a problemas de reconocimiento de patrones y optimización. Se parte de un conjunto de anticuerpos generado aleatoriamente. A partir del antígeno a neutralizar, en cada iteración se seleccionan los anticuerpos más similares, se clonan aplicando la mutación y se reintroducen en el conjunto de anticuerpos en sustitución de peores anticuerpos.

- ***Negative Selection***: Este tipo de selección se suele aplicar a problemas de detección de anomalías. Durante la fase de entrenamiento se parte de un conjunto de antígenos de entrenamiento. En cada iteración se genera un anticuerpo aleatorio, que se incorpora al conjunto de anticuerpos si no coincide con ningún elemento del conjunto de entrenamiento, o si no supera un umbral en la función de similitud. Al finalizar la fase de entrenamiento contamos con un conjunto de anticuerpos que no coinciden con los antígenos, de tal manera que al llegar un nuevo antígeno somos capaces de identificar si ya estaba en el conjunto de entrenamiento.

- ***Neighbourhood Selection***: Este tipo de selección se suele usar para sistemas de recomentación. En este caso se parte del antígeno a neutralizar y cada anticuerpo tiene un nivel de concentración. En cada iteración se añade un nuevo anticuerpo, se reducen los niveles de concentración de los anticuerpos poco similares y se aumentan los de los anticuerpos muy similares. Cuando un anticuerpo baja su concentración por debajo de un umbral, se elimina del conjunto. Este proceso se repite hasta que el sistema se estabiliza, o se alcanza la capacidad máxima de anticuerpos. Las recomendaciones se realizan a partir del conjunto de anticuerpos resultante.

Fuzzy Systems (FS)

Los sistemas difusos parten de la lógica difusa, que es una modalidad de la lógica que se basa en que el valor de verdad no tiene porque ser 1 o 0, como en la lógica booleana, sino que puede ser cualquier valor real entre 1 y 0. Al añadir esta incertidumbre, los valores de verdad y las probabilidades obtenidas nos permiten manejar conceptos subjetivos que de otro modo serían muy complicados de modelar. Otra gran ventaja de este tipo de lógica es que los resultados

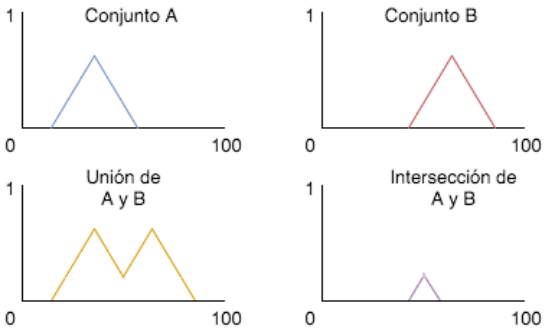


Figura 2.7: Ejemplo de unión e intersección entre dos funciones de membresía de conjuntos difusos.

obtenidos muchas veces son más fáciles de interpretar, ya que manejan conceptos difusos a los que estamos más acostumbrados.

Los sistemas difusos manejan internamente el concepto de conjunto difuso. Estos conjuntos difusos se diferencian de los conjuntos normales en que mientras que en un conjunto tradicional no hay distinción entre los distintos miembros del mismo, en un conjunto difuso si la hay, dado que cada miembro tiene un nivel de pertenencia. Por ejemplo, dos personas que miden 1.8m y 1.9m forman parte del conjunto de personas altas, pero la segunda tiene un nivel de pertenencia mayor. La definición de un conjunto difuso la marca su función de membresía, que indica el nivel de pertenencia de los distintos miembros. Por supuesto, estos conjuntos difusos tiene una serie de características distintas a los conjuntos tradicionales, y la manera de manejarlos también es distinta. Por ejemplo, en la figura 2.7 vemos los dos conjuntos difusos A y B y sus conjuntos de unión e intersección, que se definen como otra función de membresía distinta.

Es importante destacar la diferencia entre este valor de pertenencia y la probabilidad. Aunque ambos son valores reales entre 0 y 1, la probabilidad solo tiene relevancia hasta la observación del suceso, mientras que el valor de pertenencia de la lógica difusa aún tiene relevancia después. Por ejemplo en el lanzamiento de una moneda hay un 50 % de probabilidad de cara y un 50 % de probabilidad de cruz, pero una vez lanzada la moneda esta probabilidad se vuelve irrelevante porque ya se conoce si ha salido cara o cruz. Por otro lado, si determinas que una persona tiene un nivel de pertenencia al conjunto de personas altas del 90 %, y este dato se va a usar para estimar si es bueno jugando al baloncesto, tras saber si es bueno o no jugando a dicho deporte el nivel de pertenencia se mantiene y sigue siendo relevante.

Los sistemas difusos suelen usarse como controladores, ya que permiten 'controlar con frases en vez de ecuaciones'. Por ejemplo, en un sistema de control de temperatura interior de un coche, podemos indicar reglas como 'Cuando el cristal esté ligeramente empañado, si las ventanas no están cerradas del todo, poner la calefacción levemente.' o 'Si el cristal está bastante empañado, poner la calefacción al máximo'. Estas sentencias serían imposibles de manejar por un motor de reglas que no aplicase lógica difusa ya que terminos como 'ligeramente', 'levemente' o 'bastante' son inmanejables por los sistemas de computación tradicionales, mientras que si son definibles como distintos niveles de pertenencia a conjuntos difusos.

Por supuesto, para aplicar la lógica difusa en cualquier tipo de problema se requiere de una adaptación, ya que todos los sistemas utilizan datos tradicionales que hemos de convertir a los distintos valores de pertenencia que maneja la lógica difusa. Esta adaptación de los valores de entrada al sistema difuso se denomina 'fuzzificación', así como la adaptación de la salida de los mismos se denomina 'defuzzificación'. El esquema 2.8 muestra la estructura general de un sistema de control difuso.

Los inputs del sistema se convierten a valores manejables por el motor de inferencia difuso

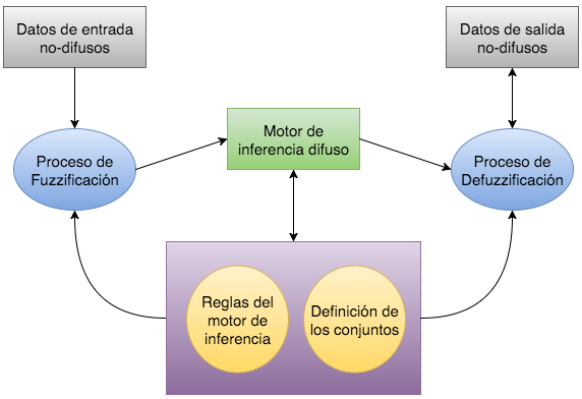


Figura 2.8: Representación básica del esquema general de un sistema de control difuso.

a través del proceso de fuzzificación ya mencionado. Esta fuzzificación define a qué conjunto difuso corresponde cada valor entrante. Una vez asignados los valores entrantes a los distintos conjuntos difusos definidos, el sistema de inferencia se encarga de aplicar las reglas establecidas. Estas reglas difusas suelen seguir el formato:

$$if \quad A\ is\ a \quad and \quad B\ is\ b \quad then \quad C\ is\ c \tag{2.3}$$

A través de la fuzzificación, el motor de inferencia sabe si $A\ is\ a$ o si $B\ is\ b$, y a través de las reglas de los conjuntos difusos, extrae conclusiones como determinar si $C\ is\ c$. Una vez extraidas las conclusiones, el sistema de defuzzificación se encarga de convertir estas conclusiones en valores útiles.

2.1.2. Inteligencia de enjambre

El concepto de Swarm Intelligence (SI), o inteligencia de enjambre, surgió en 1989 en el contexto de la robótica. Aunque no hay una definición clara, la idea es imitar el comportamiento de sistemas naturales como enjambres de insectos o bandadas de pájaros para obtener un comportamiento colectivo, descentralizado y auto-organizado de un conjunto de individuos. Cualquier sistema multi-agente y auto-organizado que muestre un comportamiento inteligente podría considerarse SI.

Un sistema SI está formado por una población de agentes simples que interactúan localmente entre si y con el entorno. Cada agente sigue reglas muy simples, y aunque no hay una estructura de control centralizada, las pequeñas interacciones locales entre los agentes hacen surgir un comportamiento global 'inteligente'. Hay numerosos algoritmos que hacen uso de estos principios: Ant Colony Optimization[6], Particle Swarm Optimization[10], Grey Wolf Optimizer[11], River Formation Dynamics[12], entre otros. En este trabajo nos centraremos en el primero de ellos, Ant Colony Optimization (ACO).

El algoritmo ACO[13, 14, 15, 16] imita el comportamiento de las colonias de hormigas. Para explicar este conjunto de algoritmos vamos a centrarnos en Simple ACO (SACO). En esta versión del algoritmo, las hormigas tratarán de encontrar el camino más corto entre dos de los nodos de un grafo (que representan el hormiguero y la comida), $G=(V,E)$, donde V es el conjunto de nodos que conforman el grafo y los enlaces, E, representan las conexiones entre los nodos.

Los enlaces del grafo contienen las feromonas depositadas por las hormigas. La cantidad de feromona depositada en la arista que conecta el nodo i con el nodo j en el *step* t se denomina $\tau_{ij}(t)$. Estas feromonas indican la calidad de los caminos que han pasado por los enlaces, de tal manera que indicarán a las hormigas que deban elegir entre distintos enlaces cuales de ellos tienen mayor probabilidad de dar lugar a una mejor solución. Inicialmente se suele suponer que no hay ninguna feromona en los enlaces del grafo, y cada hormiga que alcanza el objetivo deposita feromonas en los enlaces que ha recorrido. La cantidad de feromonas depositada es proporcional al fitness de la solución, por lo que los mejores caminos estarán más marcados, y tendrán más posibilidades de ser visitados por las siguientes hormigas.

En el primer step, N hormigas se encuentran en el nodo origen del grafo. Las hormigas se mueven a traves de los nodos contruyendo un camino (su propia solución). En cada nodo, la hormiga decide el siguiente nodo que será visitado basándose en una probabilidad. En el *step* t, dada una hormiga (k) localizada en el nodo i, la probabilidad $(p_{ij}^k(t))$ de moverse al nodo j se define por la ecuación 2.4.

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}^\alpha(t)\eta_{ij}^\beta(t)}{\sum_{u\in\mathcal{N}_i^k}\tau_{iu}^\alpha(t)\eta_{iu}^\beta(t)} & \text{if } j \in \mathcal{N}_i^k \\ 0 & \text{if } j \notin \mathcal{N}_i^k \end{cases} \tag{2.4}$$

Donde N_i^k representa el conjunto de nodos viables conectados al nodo i, para la hormiga k, mientras que η_{ij} representa el valor de la heurística al moverse del nodo i al nodo j. α es un parámetro que controla la influencia de las concentraciones de feromonas y β es otro parámetro que controla la influencia de la heurística. Estos dos parámetros sirven para encontrar un balance entre la exploración (encontrar más soluciones) y la explotación (mejorar la soluciones encontradas) del algoritmo. $\sum_{u\in\mathcal{N}_i^k}\tau_{iu}^\alpha(t)$ es el total de feronomas depositadas en todos los enlaces que conectan el nodo i con el resto de nodos.

Cuando cualquier hormiga encuentra la comida (es decir, obtiene una solución), deshace el camino de vuelta al nido depositando feromonas en el camino recorrido. La cantidad de feromonas depositadas dependerá de la calidad de la solución encontrada. Esta calidad la determina una función de fitness, que depende del problema a solucionar. Esto significa que mejores soluciones se representan con valores de feromonas más altos y estas feromonas aumentarán las probabilidades del camino de ser elegido por otras hormigas.

Por último, la cantidad de feromonas contenida en cada arista se calcula añadiendo todas las cantidades depositadas por todas las hormigas (ver ecuación 2.5).

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \sum_{k=1}^{N} \triangle\, \tau_{ij}^k(t) \tag{2.5}$$

Donde $\triangle\, \tau_{ij}^k(t)$ es la variación de las feromonas producida por la hormiga k.

Para permitir la exploración de nuevas soluciones, las feromonas sufren un proceso de evaporación. En cada iteración, las feromonas se evaporan de acuerdo a un ratio de evaporación específico. Esta evaporación se puede entender como un descenso del valor de las feromonas y se calcula siguiendo la fórmula : 2.6. Por lo tanto, el parámetro ρ, $\rho \in [0, 1]$, permite controlar como de rápido se evaporarán las feromonas. Esta evaporación sirve para evitar que aquellos caminos que representan malas decisiones sean elegidos por las hormigas.

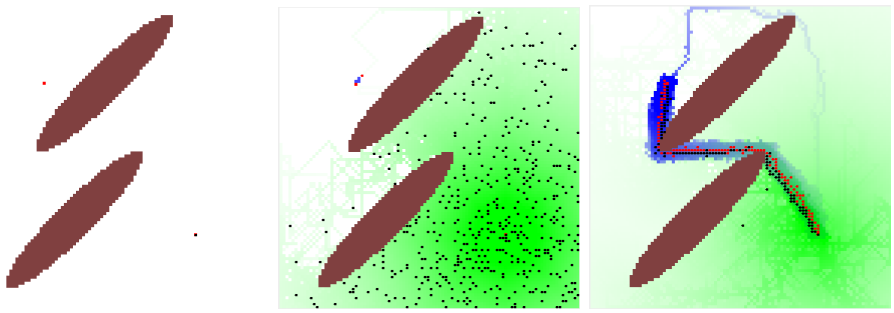
$$\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t - 1) \tag{2.6}$$

En el algoritmo 1 podemos ver el pseudocódigo de SACO. Se puede observar el uso de las distintas ecuaciones mencionadas. Hay que destacar que en la versión original de SACO, se calculaba la mejor solución como aquella cuyo recorrido era el más corto, ver línea 13. Si en ese paso incluimos el cálculo del fitness dependiente del problema, podemos aplicar algo más complejo que el camino más corto. Esto nos permite, por ejemplo, penalizar caminos que discurran por determinadas zonas, o favorecer aquellos caminos que pasen por determinados nodos del grafo.

Algoritmo 1: Simple Ant Colony Optimization

```
1  Initialize τ_ij(0) to small random values
2  Let t ← 0
3  Place n_k ants on the origin node
4  repetir
5      para cada ant k ← 1, ..., n_k hacer
6          x^k(t) ← 0
7          repetir
8              Select next node base on probability defined in equation 2.4
9              Add link (i, j) to path x^k(t)
10         hasta que destination node has been reached
11
12         Remove all loops from x^k(t)
13         Calculate the path length f(x^k(t))
14     fin
15     para cada link (i,j) of the graph hacer
16         Reduce the pheromone, τ_ij(t) using equation 2.6
17     fin
18     para cada ant k ← 1, ..., n_k hacer
19         para cada link (i, j) of x^k(t) hacer
20             Update τ_ij(t) using equation 2.5
21         fin
22     fin
23     t ← t + 1
24 hasta que stopping condition is true
25
26 Return the path x^k(t) with smallest f(x^k(t)) as the solution
```

Un ejemplo del funcionamiento del algoritmo se puede observar entre las figuras 2.9a, 2.9b y 2.9c. La figura 2.9a muestra el estado inicial de ACO. Las zonas marrones representan materiales no atravesables, el punto rojo en la zona superior izquierda representa el objetivo y el punto de la zona inferior derecha representa el nido, desde donde partirán las hormigas (puntos negros) en su búsqueda. En la figura 2.9b las hormigas han comenzado a explorar los alrededores del nido, se representan en distintas tonalidades de verde los caminos recorridos por las hormigas. Las hormigas continúan la búsqueda hasta dar con una solución. El rastro de feromonas de vuelta al nido se representará en tonalidades de azul, según su intensidad. Al continuar la exploración se acaban encontrando varios caminos al objetivo. Por último, las hormigas marcan los caminos encontrados, y se empiezan a centrar en el más eficiente de los dos, hasta encontrar la solución óptima (figura 2.9c).



(a) Estado inicial de ACO. (b) Estado intermedio. (c) Estado final.

Figura 2.9: Fases de ACO: Estado inicial del algoritmo, estado intermedio con hormigas explorando el espacio de soluciones y estado final con todas las hormigas utilizando el camino encontrado.

2.2. Computación Distribuida

En las últimas décadas, la cantidad de información que disponemos en bruto es cada vez mayor, y el modelo tradicional de computación, donde un único ordenador se encarga de procesar todo, ha quedado por completo obsoleto. A comienzos de los años 80, surgió una nueva disciplina entre las ciencias de la computación: la computación distribuida.

El objetivo de esta disciplina, junto con la computación paralela, es el procesamiento simultáneo de información entre distintas entidades denominadas 'nodos' de forma coordinada. Cada uno de estos nodos se encarga de una pequeña porción del problema global que se debe solucionar. Este trabajo conjunto y coordinado logra que los resultados superen al trabajo individual que podrían haber tenido de forma independiente, y logra también que se supere el potencial de ordenadores más potentes trabajando de forma aislada.

Las ventajas y características de este paradigma de computación son muchas. Entre otras:

- **Tolerancia a fallos**: Los sistema de computación distribuida tienen en cuenta que el hardware de cualquiera de los nodos puede fallar en cualquier momento. Estos sistemas, por lo tanto, están preparados para continuar trabajando con los nodos restantes si uno de ellos falla: sufren una penalización por la ausencia del nodo defectuoso, pero el resultado final se consigue sin más problemas.

- **Escalabilidad**: Los sistemas de computación distribuida permiten añadir y eliminar nodos rápidamente al sistema sin reiniciar ni pausar el proceso. Esto facilita enormemente aumentar la capacidad del cluster de computadores en plena ejecución cuando el problema sobrepasa las capacidades del mismo.

- **La estructura del sistema no tiene porqué ser uniforme**: Dado que cada nodo trabaja de forma independiente en su fracción del problema, las características de los distintos nodos no tienen porqué ser iguales. Se puede estar tratando con nodos que tengan distinto hardware o distinto software. Estos sistemas además se suelen adaptar a las características de los nodos con que se trabaja, por lo que es relativamente fácil reemplazar los nodos defectuosos por otros distintos o añadir más nodos al cluster sin necesidad de descartar los antiguos.

- **Perfecto para paralelizar**: Ya que cada unos de los procesos paralelos se lleva a cabo en un nodo distinto, es menos probable que se produzcan bloqueos entre los distintos procesos. Para que esto se cumpla las tareas deben estar correctamente paralelizadas.

Por supuesto, no todo son ventajas. Aprovechar al máximo este tipo de computación exige ser capaz de adaptar el problema en cuestión a este tipo de computación. Una mala división de tareas podría dar lugar a un rendimiento prácticamente igual a un procesamiento no distribuido/paralelo.

En los últimos años han surgido distintos frameworks y tecnologías que permiten trabajar con computación distribuida. Entre estos frameworks destacan los proporcionados por *Apache Software Foundation*, tanto por su calidad como por ser *open-source*: *Apache Hadoop* y *Apache Spark*.

2.2.1. Apache Hadoop

Apache Hadoop[3] es un framework de código abierto para almacenar y procesar grandes cantidades de información distribuida en clusters de ordenadores 'corrientes', entendiendo por corrientes los ordenadores que se pueden tener en casa, y no superordenadores. Fue creado en 2005 por Doug Cutting y Mike Cafarella, pero no fue hasta 2011 que se lanzó su primera versión estable.

Apache Hadoop combina su sistema de almacenamiento de información (*Hadoop Distributed File System* o HDFS) con su sistema de procesamiento basado en *MapReduce*. El primero de ellos divide archivos en grandes bloques que distribuye entre los nodos del cluster, encargándose de que cada uno de estos nodos tenga disponible la información necesaria para llevar a cabo sus tareas. Una vez que la información se encuentra distribuida, *Hadoop MapReduce* transfiere código empaquetado a cada uno de estos nodos, dependiendo de los datos que deba procesar el nodo. Esta aproximación permite aprovechar que los datos a procesar se encuentran distribuidos localmente, ya que cada uno de los nodos trata únicamente con la porción de datos que le ha sido asignada y distribuida previamente.

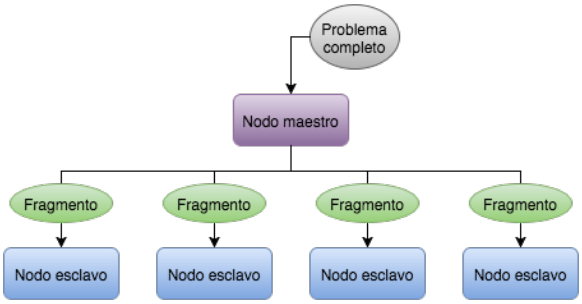


Figura 2.10: Esquema general de un cluster de ordenadores de *Apache Hadoop*

La arquitectura general de *Apache Hadoop* se puede observar en la figura 2.10. *Hadoop* cuenta con un nodo maestro y multiples nodos esclavos. El nodo maestro contendrá toda la información inicialmente, y se encargará de distribuir tanto la información como las tareas entre los distintos nodos esclavos. Mientras que el nodo maestro debe tener algo más de capacidad, los nodos esclavos basta con que puedan ejecutar cada tarea individualmente en un momento concreto. Si las tareas se han analizado e implementado correctamente, las características mínimas de los nodos esclavos son relativamente bajas. Esto permite que obtener más nodos sea más económico, permitiendo aumentar el sistema fácilmente.

Apache Hadoop ha sido muy usado por grandes compañias. Yahoo! comenzó usándolo en 2008, seguido de Facebook en 2010. En 2013 más de la mitad de las compañías de Fortune 50 hacían uso de *Apache Hadoop*. Además, muchas compañías de procesamiento en la nube ofrecen facilidades para el uso de *Hadoop*. *Microsoft Azure*, *Amazon EC2*, *Google Cloud Platform*... todos ofrecen métodos fáciles y rápidos de desplegar *Hadoop* en sus clusters de ordenadores.

MapReduce

En esta subsección veremos el paradigma *MapReduce*, que es una parte fundamental tanto de *Apache Hadoop* como de *Apache Spark*. *MapReduce* es un modelo de programación especialmente implementado para tratar con grandes cantidades de datos aplicando algoritmos paralelizados y distribuidos. Aunque fue Google quien desarrolló originalmente *MapReduce*, actualmente el término se aplica genéricamente.

Siguiendo el paradigma 'Divide y vencerás', *MapReduce* se compone tradicionalmente de una operación *Map* y una operación *Reduce*. Adaptar un algoritmo a estas dos operaciones que explicaremos a continuación permite su correcta distribución y que pueda ser ejecutado en un cluster de ordenadores.

La idea principal se muestra en la figura 2.11: se divide el problema obteniendo pares clave-valor, se aplica *Map* a cada uno de estos pares y después se obtiene la solución aplicando *Reduce* a los resultados del procesamiento de *Map*. Al paralelizar estas operaciones, cada nodo que deba realizar una tarea necesita una porción muy pequeña del problema total. De hecho, se puede observar cómo no es necesario que se completen todas las operaciones *Map* para comenzar con las operaciones *Reduce*.

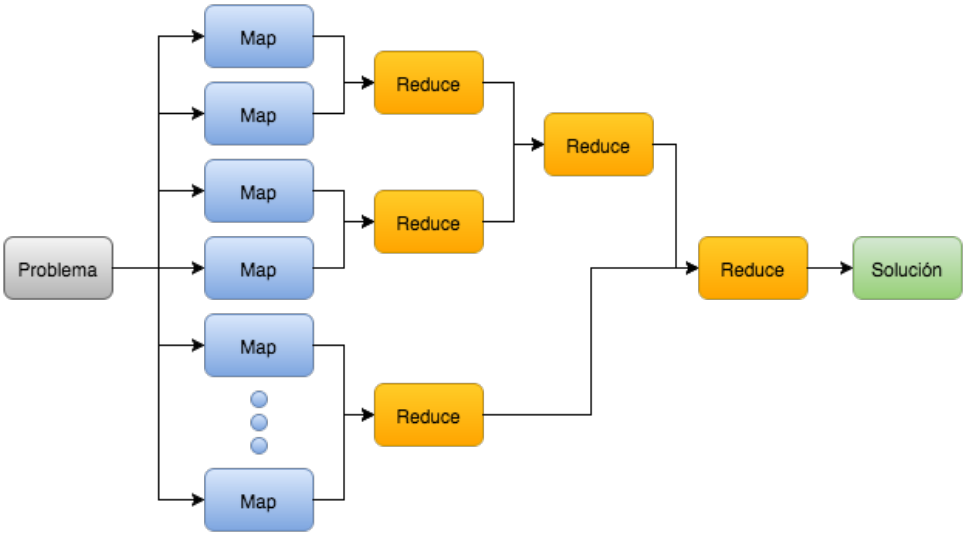


Figura 2.11: Esquema general de *MapReduce*

Tradicionalmente, la operación *Map* suele aplicar funciones de filtro y ordenación, mientras que la función *Reduce* aplica funciones de recolección como contar elementos o seleccionar únicamente los indicados.

El ejemplo más clásico de *MapReduce* se puede observar en la figura 2.12 donde se muestran cómo se puede contar las palabras de un texto usando las dos operaciones ya descritas. Al dividir la información, cada operación *Map* se encargaría de un párrafo. La operación *Reduce* se encargaría de sumar los resultados obtenidos.

MapReduce y Apache Hadoop

Apache Hadoop permite implementar todos los pasos que componen *MapReduce* desde la lectura del fichero de origen de los datos hasta la escritura de los resultados de nuevo en fichero, aunque cuenta con implementaciones por defecto.

Los pasos que se pueden implementar son los siguientes: la lectura del fichero, la aplicación

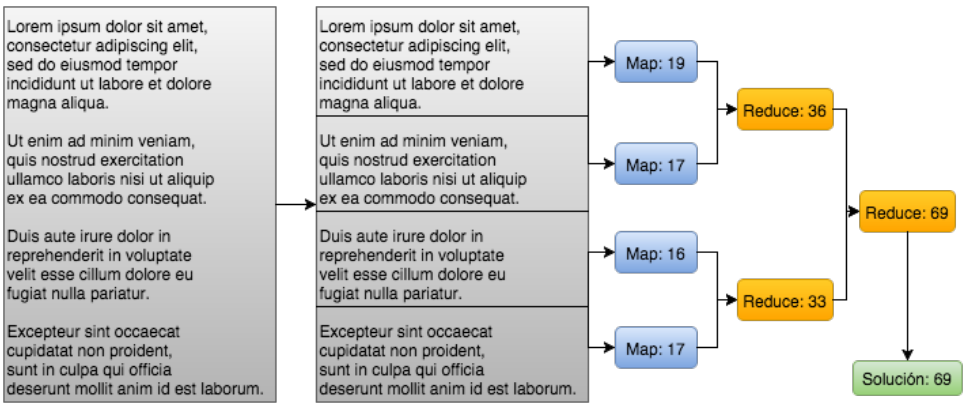


Figura 2.12: Ejemplo de aplicación de MapReduce

de la función *Map*, la distribución de los resultados entre los *Reduce*, la función *Reduce* y la escritura del resultado en fichero.

La lectura del fichero incluye tanto la lectura física del fichero como su conversión en pares clave-valor. La implementación por defecto divide el fichero en fragmentos manejables de entre 64MB y 128MB, y convierte cada línea en un par clave-valor donde la clave es el número de línea y el valor es el contenido de la misma. Gracias a las implementaciones que nos permite hacer *Hadoop*, podemos interpretar el contenido de los ficheros de la manera que gustemos antes incluso de comenzar a aplicar *Map*, añadiendo un preprocesamiento a los datos que puede acelerar el manejo de los mismos posteriormente. Por ejemplo, en una implementación de Ant Colony Optimization, podríamos leer la configuración deseada de las hormigas a ejecutar en un fichero y convertirlas en pares (Número de hormiga, Configuración).

La aplicación de la función *Map* nos permite implementar el procesamiento de los pares clave-valor. La función *Map* no tiene porqué procesar un único par, como tampoco tiene porqué devolver un único par. Por ejemplo, una única hormiga de ACO obtiene una única solución, pero podría convenirnos implementar una función *Map* personalizada que recibiera más de una configuración y devolviera más de una solución encontrada. O podría no devolver ninguna solución en caso de no encontrarse.

La distribución de los resultados se encarga de decidir a qué nodo *Reduce* se asignará qué par. Normalmente el objetivo de esta distribución es balancear la carga para compensar la presencia de algún nodo *Reduce* especialmente lento, y al mismo tiempo mezclar los resultados de *Map*. También se puede indicar el orden en que se quiere que se procesen los pares asignados a un nodo *Reduce*.

La función *Reduce* se aplica una vez por cada clave única del conjunto de pares clave-valor que se le haya asignado. Esta función recibirá el conjunto de pares que compartan dicha clave para producir un resultado, y devolverá un conjunto de pares clave-valor procesadas. Esto implica que debe tenerse muy en cuenta la clave asignada en la función *Map*. En el ejemplo de ACO que hemos mencionado antes, podemos buscar que todas las soluciones tengan una clave común, provocando que todas se procesen en un único *Reduce*, dando lugar a un único conjunto de soluciones. O podríamos dar a cada configuración distinta ejecutada una clave distinta, por lo que las soluciones de cada configuración se procesarían en *Reduce* distintos, agrupando las soluciones encontradas según su configuracion.

Por último, se puede configurar como guardar los resultados producidos. Un correcto formateo en este último paso facilita enormemente la interpretación y el uso de los datos resultantes.

Resumiendo este apartado, aunque la aproximación *MapReduce* se encuentra acotada por las limitaciones del proceso que hemos explicado, *Hadoop* permite personalizar todos los pasos.

2.2.2. Apache Spark

La gran desventaja de *Apache Hadoop* es que todas sus operaciones hacen uso de HDFS, por lo tanto de acceso a disco. *Apache Spark* soluciona esto, haciendo todas las operaciones en memoria y pudiendo llegar a operar hasta 100 veces más rápido que *Apache Hadoop*. *Apache Spark* es muy reciente, comenzó a desarrollarse en la universidad de Berkeley en 2009 y su primera versión estable salió en 2014. Actualmente sigue en desarrollo. Además, a dia de hoy cuenta con el record mundial de ordenación de grandes cantidades de datos, obtenido en Noviembre de 2014. Al igual que *Apache Hadoop*, los grandes proveedores de servicios de computación en la nube facilitan el uso de *Spark* en sus clusters de computadores.

Además de las operaciones *Map* y *Reduce* que *Apache Hadoop* ofrece, *Apache Spark* incluye una nueva estructura de datos: *Resilient Distributed Datasets* o RDD. Todas las operaciones de *Apache Spark* se realizan sobre RDDs. Los RDDs ofrecen un nuevo nivel de abstracción para los datos utilizados que permite manejar de forma mucho más cómoda y eficiente el problema que se pretende solucionar. Estos RDDs se puede obtener a partir de datos de sistemas de almacenamiento externos o aplicando operaciones sobre otros RDDs: map, reduce, join, filter...

La facilidad de uso los RDDs favorece la aplicación de *Spark* a pequeñas operaciones que a través de *Apache Hadoop* serían demasiado complejas de llevar a cabo. Gran parte de esta facilidad de uso se debe a la integración de estos RDDs en los lenguajes Java, Python, Scala y R, que permite su uso de forma similar a las colecciones y listas nativas de estos lenguajes. En el caso de Python, el lenguaje utilizado en este Trabajo de Fin de Master, se acentúa la flexibilidad de *Spark* al ser un lenguaje con tipado dinámico y contar con funciones anónimas o *lambdas* para reducir la verbosidad del código, como veremos más adelante en el ejemplo.

Por otro lado, *Apache Spark* incluye un framework de aprendizaje automático distribuido llamado *MLlib*. *MLlib* aprovecha la arquitectura de *Spark* y su distribución de datos en memoria para mejorar enormemente las implementaciones anteriores basadas en almacenamiento en disco de *Apache Mahout* y *Vowpal Wabbit*, tanto en velocidad como escalabilidad. Este framework cuenta con gran cantidad de algoritmos estadísticos y de aprendizaje automático que facilitan su uso y su integración en operaciones más complejas, incluyendo:

- Resumen de estadísticas, correlación y generación aleatoria de datos.
- Clasificación y regresión: regresión logística, regresión lineal, árboles de decisión, clasificación por Naive Bayes, Support Vector Machines (SVM)...
- Métodos de clustering como k-means y Latent Dirichlet Allocation (LDA).
- Técnicas de reducción de dimensionalidad como Singular Value Decomposition (SVD) o Principal Component Analysis (PCA).

Cabe destacar que *MLlib* no cuenta con ninguna implementación de *Ant Colony Optimization* ni *Ant Colony Optimization* para tareas de Clustering, lo que es una de las motivaciones de este trabajo.

Entendiendo Apache Spark

En este apartado veremos un ejemplo de uso de la API de *Apache Spark* en Python: *PySpark*. Se aprecia rápidamente la facilidad de uso del modelo de programación *Spark* y su potencial.

En el fragmento de código 2 se puede observar el ejemplo tradicional de computación distribuida, contar el número de veces que aparece cada palabra en un texto. Tras iniciar *Spark* en

la línea 4, se carga el fichero en la línea 5, obteniendo un RDD que contiene cada una de las lineas del fichero por separado y sin procesar. El primer procesamiento que se lleva a cabo es la función *flatMap* en la línea 6, que aplica la función indicada y concatena los arrays resultantes. En este caso se usa para dividir cada linea en arrays de palabras y concatenar los resultados, obteniendo un único array con todas las palabras del texto. Posteriormente se aplica *map* en la línea 7, creando pares clave-valor usando como clave la palabra y como valor el número de veces, inicialmente 1. Tras ello, en la línea 8 *reduceByKey* se encarga de aplicar la función indicada, en este caso *add*, sobre los pares palabra-valor tras haber sido agrupados por clave. Es decir, que suma el número de veces que aparece cada palabra. Por último, la función *collect* se encarga de obtener los resultados de las operaciones anteriores.

Algoritmo 2: WordCount con PySpark

```
1  import sys
2  from operator import add
3  from pyspark import SparkContext
4  sc = SparkContext()
5  lines = sc.textFile('text.txt')
6  counts = lines.flatMap(lambda x: x.split(' '))
7      .map(lambda x: (x, 1))
8      .reduceByKey(add)
9  output = counts.collect()
10 for (word, count) in output:
11     print word + ':' + str(count)
12 sc.stop()
```

Como se puede observar, la carga de un fichero con *sc.textFile* produce un RDD. Es decir, una serie de datos guardados en memoria y preparados para ser distribuidos entre los nodos a la hora de paralelizar. Cada una de las funciones que aplicamos, *flatMap*, *map*, *reduce*, etc, produce un nuevo RDD sobre el cual se podrán llevar concatenar más operaciones hasta finalizar con un *recolector*. Estos recolectores, como el *collect* que hemos usado o *max* o *min*, se encargan de procesar todos los pasos anteriores y obtener el resultado en una variable nativa del lenguaje utilizado.

Hay que tener en cuenta que hasta que no se invoca un recolector, no se realiza el procesamiento. Es decir, *counts* no contiene un conjunto de pares (palabra, número de apariciones), sino que contiene la secuencia de funciones que lleva a ese resultado. Es en la línea 9 donde se llevan a cabo las operaciones al invocar al recolector *collect*. Esto permite reducir la memoria consumida, dado que en el ejemplo visto no se ha duplicado la información de la variable *lines* en ningún momento. Tambien facilita la comprensión del código, ya que podemos permitirnos asignar los resultados parciales de las operaciones a variable sin influir en el rendimiento del proceso.

Spark cuenta tambien con el uso de *Broadcast Variables*, que permiten acceder a información inmutable desde el interior de la computación. En el ejemplo 3 vemos un uso sencillo de estas variables al asignar el peso de cada palabra obteniendolo del exterior de la función.

Otra de las grandes ventajas de *Spark* es que estos fragmentos de código serán exactamente los mismos independientemente de si se despliegan sobre un cluster de 8 nodos o de 80, o incluso de si se despliegan en un ordenador paralelizando entre los núcleos del procesador.

Algoritmo 3: WordCount con PySpark haciendo uso de *Broadcast Variables*

```
1  import sys
2  from operator import add
3  from pyspark import SparkContext
4  sc = SparkContext()
5  lines = sc.textFile('text.txt')
6  broadcast = sc.broadcast(1)
7  counts = lines.flatMap(lambda x: x.split(' '))
8      .map(lambda x: (x, broadcast.value))
9      .reduceByKey(add)
10 output = counts.collect()
11 for (word, count) in output:
12     print word + ':' + str(count)
13 sc.stop()
```

3
Implementación

En este capítulo comenzaremos analizando el algoritmo ACOC que se implementará en el apartado 3.1. Después veremos como se ha adaptado este algoritmo al *framework Spark* en los apartados 3.2.2 y 3.2.3.

3.1. Descripción de ACOC

El algoritmo *Ant Colony Optimization* para tareas de Clustering (ACOC)[7] tiene como principal objetivo aplicar las ventajas de ACO a problemas de clusterización. En ACOC, el espacio de soluciones se representa como una matriz de nodos objeto-cluster, con un número m de filas correspondiente con el número de objetos a clusterizar y un número g de columnas correspondiente al número de clusters que se desean. Cada nodo $N(i, j)$ indicaría que el objeto i ha sido asignado al cluster j. Las hormigas solo pueden seleccionar uno de los g nodos para cada objeto, lo que significa que cada elemento sólo puede pertenecer a un único cluster. La figura 3.1 muestra como se seleccionan los distintos clusters para cada uno de los nodos.

En este grafo, las hormigas se desplazan de un nodo a otro construyendo una solución. En cada paso la hormiga selecciona un objeto no clasificado y lo añade a la solución en función de la intensidad de las feromonas y de la información heurística. La solución se considera completa cuando todos los objetos han sido clusterizados. Una vez encontrada dicha solución, se depositan feromonas en los nodos seleccionados dependiendo de la calidad de la misma: a mayor calidad, mayor cantidad de feromonas. Esta calidad se evalua midiendo las distancias Euclideas de cada objeto al centro del cluster que le ha sido asignado. Los nodos con mayor cantidad de feromonas tienen más probabilidad de ser seleccionados.

3.1.1. Algoritmo ACOC

A continuación describiremos los parámetros, valores internos y puntos clave del algoritmo.

Los parámetros del algoritmos son los siguientes:

- m: Número de objetos a clusterizar.

- n: Número de atributos de cada objeto.
- g: Número de clusters deseados.
- R: Número de hormigas.
- r: Número de hormigas elitistas que actualizarán feromonas.
- I: Número de iteraciones.
- ρ: Tasa de evaporación
- β: Influencia de la heurística
- α: Influencia de las feromonas
- q_0: Umbral de explotación.

Este algoritmo maneja los siguientes valores internos:

- **PM**: Matriz $m \times g$ de feromonas.
- **tb**: Lista de nodos ya clusterizados. Propio de cada hormiga.
- **C**: Matriz $g \times n$ de centros de los clusters. Propio de cada hormiga.
- **W**: Matriz $m \times g$ de pesos, indica con 1's y 0's que objetos se han asignado a que clusters. Propio de cada hormiga.

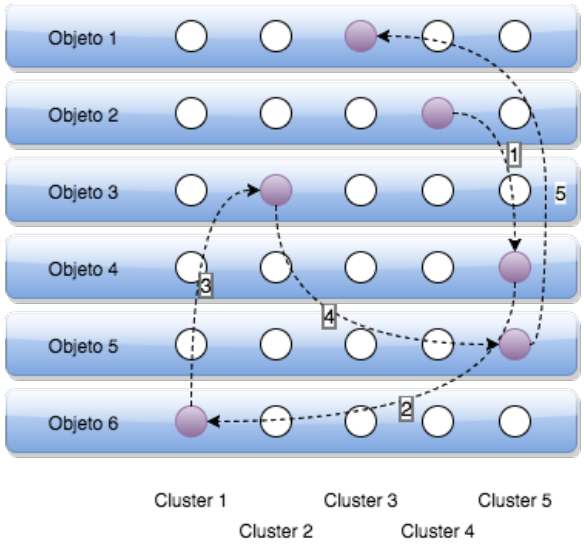


Figura 3.1: Grafo a recorrer por nuestras hormigas

Inicialización de PM

La inicialización de **PM**, en la línea 1, se realiza seleccionando valores aleatorios pequeños. Es necesario inicializar las feromonas con valores pequeños para el correcto funcionamiento del algoritmo. Dado que se trata de valores muy pequeños, el aspecto estocástico de esta inicialización apenas influye en el resultado final.

Algoritmo 4: Pseudocódigo del algoritmo ACOC

```
1  Initialize PM
2  iteration ← 0
3  mientras iteration < I hacer
4      para todo R ants hacer
5          reset tb
6          reset C
7          reset W
8          mientras tb is not full hacer
9              i ← selectObject(tb)
10             c ← selectCluster(i, C, PM)
11             updateInformation(W, C, tb, i, c)
12         fin
13     fin
14     updatePM(PM)
15     iteration + +
16 fin
17 return best solution
```

Selección de objeto

El siguiente objeto a clusterizar se selecciona en la línea 9 de una forma completamente aleatoria de la lista de objetos restantes a clusterizar. Cada objeto se seleccionará una única vez, dado que un objeto no puede encontrarse en dos clusters a la vez.

Selección de cluster

Una vez seleccionado el objeto i a clusterizar, la selección del cluster, en la línea 10, que le corresponde es más compleja, dado que corresponde con el procedimiento fundamental de la hormiga. Por un lado, tenemos una estrategia de explotación influida por las feromonas. Por otro lado, tenemos una estrategia de exploración determinado por la heurística. En ambas estrategias el parámetro β regula el peso de la heurística y α el peso las feromonas.

Se parte de la base de que todos los nodos candidatos tienen una puntuación correspondiente al valor de las feromonas (τ) multiplicado por el valor de su heurística (η), ver ecuación 3.1.

$$p(i,j) = [\tau(i,j)]^{\alpha}[\eta(i,j)]^{\beta} \tag{3.1}$$

Una probabilidad aleatoria q determina si se aplica la estrategia de explotación ($q <= q_0$) o la de exploración ($q > q_0$).

La estrategia *greedy* selecciona aquel cluster cuyo nodo tenga la mejor puntuación, ver ecuación 3.2.

$$j = argmax_{j \in g}\{p(i,j)\} \tag{3.2}$$

Por otro lado, la estrategia de exploración realiza una selección aleatoria proporcional a la puntuación de los nodos candidatos. La probabilidad de cada nodo se calcula con la ecuación 3.3.

$$P(i,j) = \frac{p(i,j)}{\sum_{0<u<g} p(i,u)} \tag{3.3}$$

Actualización de información

En la actualización de información se realizan las siguientes operaciones:

- Se actualiza **W**, marcando como 1 el nodo seleccionado.
- Se actualiza **C**, recalculando el centro del cluster seleccionado.
- Se actualiza **tb**, añadiendo el nodo a la lista de nodos visitados para no repetirlo.

Actualización de matriz de feromonas

Una vez se tienen todas las soluciones de todas las hormigas, se evaluan y se ordenan, para depositar las feromonas de las r mejores.

La calidad de una solución se calcula con la fórmula 3.4. Donde w_{ij} define si el nodo ha sido seleccionado y $d(X_i, C_j)$ es la distancia entre un nodo y el centro de su cluster.

$$J(W,C) = \sum_{i=1}^{m}\sum_{j=1}^{g} w_{ij} \cdot d(X_i, C_j) \tag{3.4}$$

La actualización de feromonas del nodo $N(i,j)$ se realiza siguiendo la fórmula 3.5.

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \frac{1}{J(W,C)} \tag{3.5}$$

Por último se realiza una evaporación de todas las feromonas de la matriz con la fórmula 3.6.

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) \tag{3.6}$$

Cálculo de la heurística

El cálculo de la heurística de la pertenencia de un nodo a un cluster se realiza calculando la distancia al centro del mismo mediante la fórmula 3.7.

$$d(X,C) = \sqrt{\sum_{v=1}^{n}(X_v - C_v)^2} \tag{3.7}$$

3.2. Paralelización de ACOC

El objetivo de adaptar ACOC a Spark es aprovechar las capacidades de Spark para computación paralela en multiples nodos. Para explotar al máximo las ventajas que esta modalidad de computación ofrece, el algorimo debe adaptarse y maximizar su paralelización.

Hay múltiples posibilidades de paralelización para un algoritmo ACO. En este trabajo se busca optimizar el algoritmo ACOC, no dividir el problema en subproblemas ni métodos similares que también valdrían para distribuir la carga de trabajo.

En primer lugar, vamos a analizar dónde introducir la paralelización. Después analizaremos más en detalle cómo llevar a cabo esta paralelización con Spark y por último analizaremos diferentes alternativas para optimizar el proceso.

3.2.1. Identificación de los puntos de paralelización

En el algoritmo 5 vemos el funcionamiento simplificado de ACOC con las principales operaciones del proceso. De estas operaciones, la selección del cluster para cada objeto es la más costosa, ocupando más de un 75 % del tiempo total de la ejecución.

Algoritmo 5: Pseudocódigo simplificado del algoritmo ACOC

```
1  initialization()
2  para cada iterations hacer
3      para todo ants hacer
4          mientras objectsNotClustered hacer
5              selectObject()
6              analyzeHeuristicValues()
7              updatePersonalSolution()
8          fin
9      fin
10     processSolutions()
11 fin
12 return best solution
```

No buscamos paralelizar el bucle exterior (línea 2), ya que es necesario que sea secuencial para el correcto funcionamiento del algoritmo. De hecho, a partir de ahora ni consideraremos el bucle exterior, para poder referirnos más cómodamente a los distintos pasos del algoritmo.

El segundo bucle (línea 3), que se corresponde con las hormigas que recorren el grafo, es perfectamente paralelizable: las ejecuciones son completamente independientes entre sí y el resultado de todas las ejecuciones es necesario solo cuando ya han terminado todas, como muestra la figura 3.2. En este caso se lanzarían en paralelo R procesos Spark, uno por cada hormiga. Cada uno de estos procesos recibirían como parámetros la lista de objetos y los parámetros de configuración (ρ, β, q_0...) y devolvería una solución.

Por último, el bucle más interno es secuencial, pero al contener la operación más pesada(el análisis de heurísticas), parece razonable su paralelización. La primera opción para paralelizar este bucle interno es paralelizar tanto las hormigas como la selección de los clusters. En este caso tendríamos tareas retroalimentadas, de tal manera que los procesos recibirían además **W**, **C** y **tb** para saber el estado de la hormiga. El proceso seleccionaría el cluster del siguiente objeto, actualizando **W** y **C** y eliminándo el objeto clusterizado de **tb**. El resultado del proceso se reincorporaría al pool de tareas pendientes. Este ciclo continuaría hasta haber acabado **tb**, como muestra la figura 3.3. Por desgracia ese bucle de reincorporar las tareas al pool de tareas pendientes no es posible realizarlo con *Spark* cuando el número de objetos a clusterizar es muy elevado, por lo que esta posibilidad queda descartada.

La segunda opción para paralelizar el bucle interno es paralelizar el proceso interno de *selectObjectAndCluster*(), que como ya hemos mencionado ocupa el 75 % del tiempo de procesamiento. Esta búsqueda del objeto y su cluster contiene interiormente gran cantidad de bucles

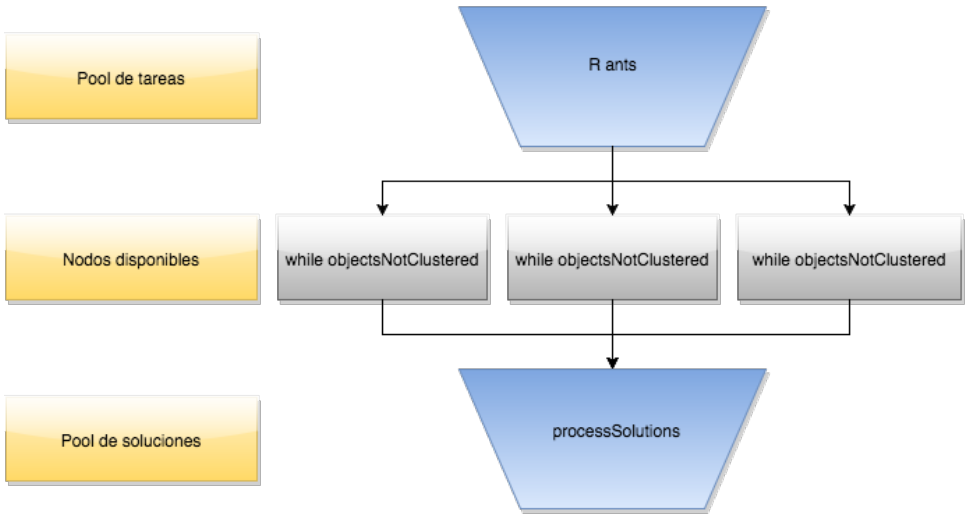


Figura 3.2: Paralelización de las hormigas de una iteración

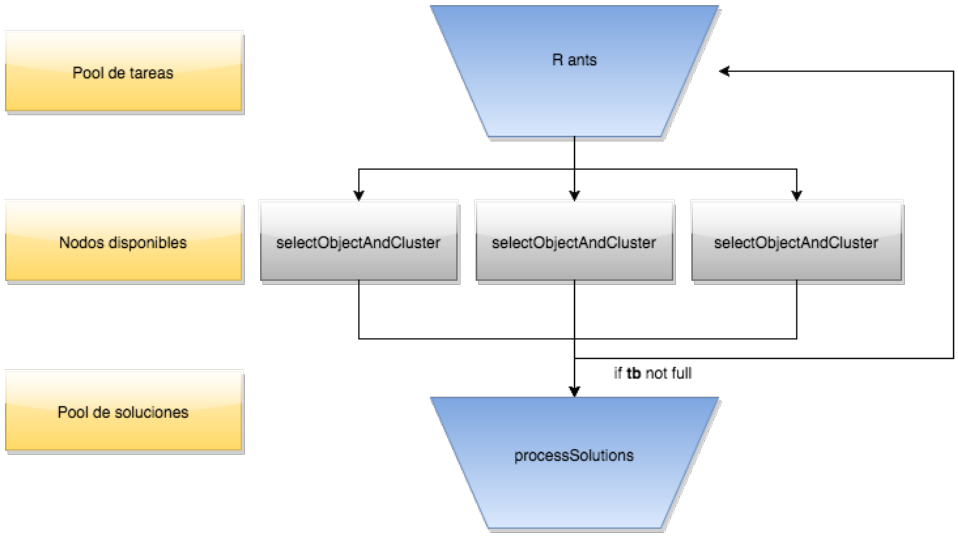


Figura 3.3: Paralelización de la selección de cluster y las hormigas simultaneamente

que recorren las distintas matrices del algoritmo, por lo que la optimización de este proceso podría mejorar considerablemente los tiempos de ejecución del algoritmo.

3.2.2. Modelo 1: paralelización de las hormigas

Una vez identificados los puntos de paralelización, procedemos a analizar cómo realizar estas paralelizaciónes con *Spark*. *Spark* no permite lanzar procesos *Spark* desde el interior de una de sus tareas, por lo que no se pueden paralelizar dos bucles anidados a la vez. Dado que no podemos paralelizar a la vez las hormigas y la selección de cluster, comenzaremos analizando la paralelización de las hormigas.

Los aspectos importantes para la paralelización de cualquier algoritmo en *Spark* requieren de la definición de las *Broadcast variables* y las subtareas que compondrán el proceso.

Broadcast variables

Las *Broadcast variables* son variables solo de lectura accesibles desde todos los nodos y tareas de un proceso *Spark*. Permiten acceder a la información rápidamente sin tener que distribuirla con el proceso que la necesita. Estas variables permiten aligerar la distribución de tareas entre los distintos nodos y por tanto aceleran el proceso en conjunto. Vamos a analizar los datos o estructuras que son comunes a todas las operaciones, y que por tanto es innecesario transportar en cada tarea.

Estos datos los guardaremos como *Broadcast variables*, de tal manera que todas las tareas tengan acceso a ellas de la manera más rápida posible. Los datos comunes que no serán modificados en nuestras tareas son los siguientes:

- Parámetros de configuración: m, n, g, R, r, l, ρ, β, α, q_0
- Objetos a clusterizar
- La matriz de feromonas **PM**

De estas *Broadcast variables*, los parámetros de configuración y los objetos a clusterizar son inmutables durante todo el algoritmo, por lo que bastaría distribuir esos datos en una única ocasión durante la inicialización de ACOC. Por otro lado, **PM** es necesario que se redistribuya como *Broadcast variables* en cada iteración, dado que el valor existente en el instante 't' se habrá actualizado con las soluciones encontradas en 't+1'.

Análisis de subtareas

En el análisis del punto de paralelización hemos determinado que en este caso la paralelización se realizará sobre las hormigas. Necesitaremos entonces una tarea *Spark* que reciba los parámetros de configuración, los objetos a clusterizar y la matriz de feromonas y devuelva una solución. Como acabamos de ver, todos estos datos serán *Broadcast variables*, por lo que nuestra tarea *Spark* individual tendrá un contexto de ejecución bastante ligero.

A partir de estos datos, nuestra tarea *Spark* pondrá a la hormiga a recorrer el grafo hasta haber construido una solución completa, asignando cada uno de los objetos a los distintos clusters. Aprovechamos esta tarea para evaluar la solución obtenida, de tal manera que tenemos una solución y además conocemos su calidad. Una vez tenemos el conjunto de soluciones obtenidas, falta seleccionar las mejores y actualizar la matriz de feromonas.

La selección de las mejores puede realizarse una vez más a través de *Spark*. *Spark* proporciona una función *takeOrdered* que permite obtener las r mejores soluciones de entre las obtenidas con la tarea anterior.

Por último, la actualización de las feromonas se realizará de forma no paralela, ya que solo debe realizarse una única vez y además el resultado debe redistribuirse como *Broadcast variable* machacando lo anterior. El esquema final será como muestra la figura 3.4.

3.2.3. Modelo 2: paralelización de la selección de cluster

Como ya se ha mencionado,*Spark* no permite paralelizar en el contenido de una tarea *Spark*, por lo que en este apartado analizaremos como paralelizar la selección de objeto y su asignación a un cluster.

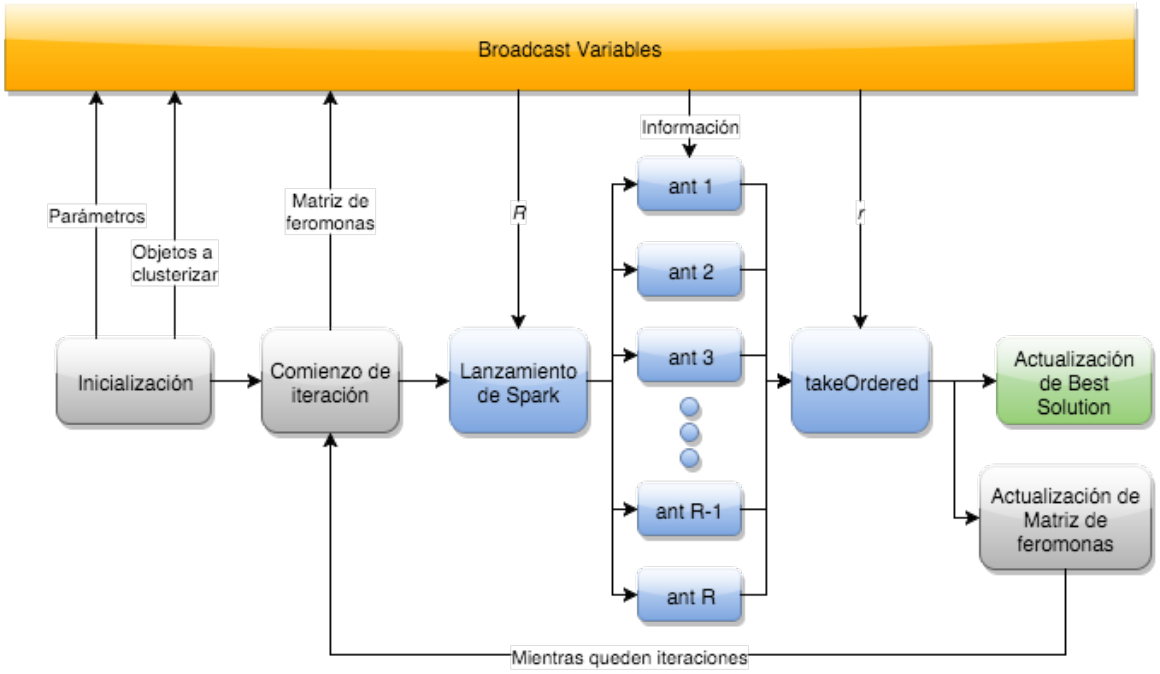


Figura 3.4: Esquema total del proceso habiendo paralelizado las hormigas

La selección del objeto, como ya hemos visto en la definición del algoritmo ACOC, es aleatoria. Se trata de la asignación a un cluster la operación compleja que hemos de optimizar.

En la definición de ACOC hemos visto que para seleccionar un cluster en primer lugar se debe calcular la puntuación de cada cluster con la fórmula 3.8. Posteriomente, y en función de un valor aleatorio q, se determina si se aplicará la estrategia *greedy* (fórmula 3.9) o la estrategia de exploración (fórmula 3.10).

$$p(i,j) = [\tau(i,j)]^{\alpha}[\eta(i,j)]^{\beta} \tag{3.8}$$

$$j = argmax_{j \in g}\{p(i,j)\} \tag{3.9}$$

$$P(i,j) = \frac{p(i,j)}{\sum_{0<u<g} p(i,u)} \tag{3.10}$$

Como conclusión, la asignación de un cluster sigue el algoritmo 6, donde $calculateClusterValue$ depende de la distancia del centro del cluster al objeto y de la cantidad de feromonas de la elección objeto-cluster. El centro del cluster se encuentra calculado de antemano, por lo que nos ahorramos otro bucle adicional para calcularlo.

El objetivo de esta segunda paralelización será optimizar y acelerar el cálculo del valor de cada cluster.

Una vez más, los puntos importantes a establecer son las *Broadcast variables* y las subtareas que compondrán el proceso.

Broadcast variables

Como ya hemos mencionado, las *Broadcast variables* son variables solo de lectura accesibles desde todos los nodos y tareas de un proceso *Spark*. En este caso, los datos comunes que no es necesario modificar en nuestras tareas son los siguientes:

Algoritmo 6: Pseudocódigo de selección de un cluster para un objeto

```
1 obj ← selectRandomObject()
2 para todo cluster hacer
3 |  calculateClusterValue(cluster, obj)
4 fin
5 si exploración entonces
6 |  return weightedChoice()
7 en otro caso
8 |  return bestCluster()
9 fin
```

- Parámetros de configuración: m, n, g, R, r, l, ρ, β, α, q_0
- Objetos a clusterizar
- La matriz de feromonas **PM**
- Centros de los clusters

Al igual que en el caso anterior, los parámetros de configuración y los objetos a clusterizar son inmutables durante todo el algoritmo, por lo que solo los distribuiremos en una única ocasión: durante la inicialización de ACOC. La matriz de feromonas, al igual que en el modelo anterior, solo variará al finalizar todas las hormigas, por lo que solo es necesario actualizarla al comienzo de cada iteración. En el caso de los centroides, éstos son necesarios para los cálculos de la tarea a paralelizar, pero no se ven modificados en su interior por lo que deben ser distribuidos inmediatamente antes de la paralelización.

Análisis de subtareas

En este segundo modelo, necesitaremos una tarea *Spark* que reciba los parámetros de configuración, los objetos a clusterizar, la matriz de feromonas y el centro del cluster y devuelva el valor de ese cluster para el objeto indicado. La mayor parte de estos datos serán *Broadcast variables*, por lo que nuestra tarea *Spark* individual únicamente recibirá como argumentos el objeto y el cluster para los cuales debe obtener el valor.

Dado que el nivel de paralelización en este caso es a un nivel mucho más específico que la paralelización de las hormigas, las tareas *Spark* únicamente aplicarán la fórmula 3.8 y devolverán el resultado obtenido.

El esquema final de esta segunda versión del algoritmo paralelizando la selección de los clusters será como muestra la figura 3.5. La paralelización se encuentra en el interior de cada hormiga, paralelizando el cálculo de los valores heurísticos de los distintos clusters para cada objeto a clasificar.

3.2.4. Optimización del manejo de datos

Uno de los objetivos es poder aplicar este algoritmo y su implementación a grandes cantidades de datos. Esto nos exige que tratemos de optimizar al máximo posible el consumo de memoria, ya que al ampliar el volumen de datos éste se acaba convirtiendo fácilmente en un cuello de botella. Parte de las optimizaciones ya las hemos llevado a cabo a través del uso eficiente de las *Broadcast variables*, pero al margen de esto hay una serie de consideraciones importantes:

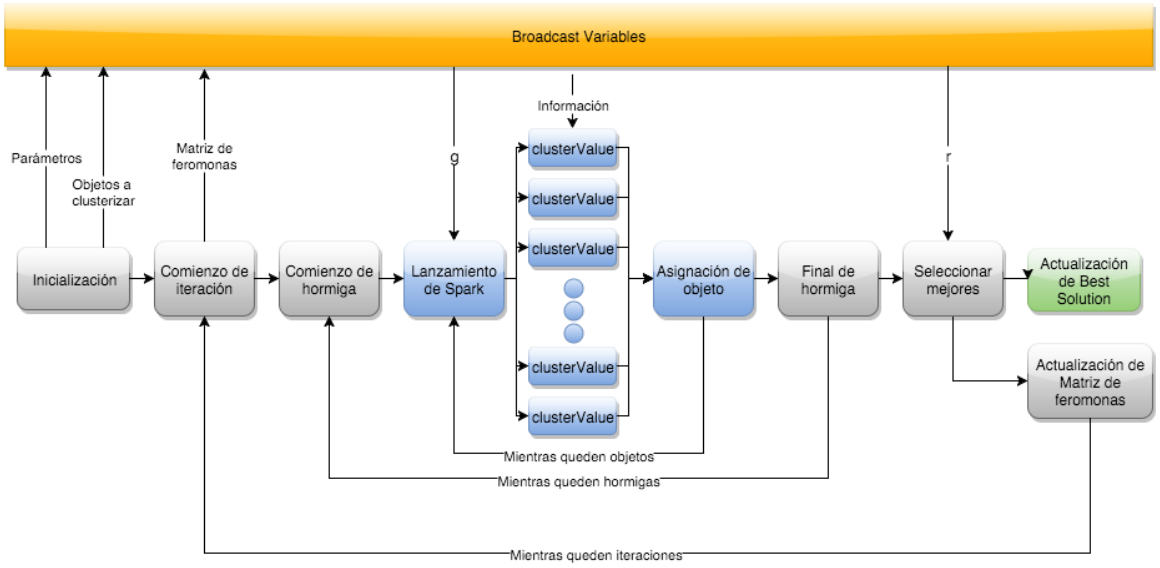


Figura 3.5: Esquema total del proceso habiendo paralelizado la selección de cluster

- La matriz de pesos, **W**, que indica con 1's y 0's qué objetos se corresponden con qué clusters, contiene mucha información innecesaria. Esto se debe a que se trata de una matriz *sparse*, donde solo nos interesan los 1's que ocupan un $\frac{1}{g}$ del total de la matriz, siendo g el número de clusters. Es mucho más adecuado guardar el cluster seleccionado para cada uno de los objetos, a cambio de adaptar las operaciones correspondientes que tienen que ver con **W**. Esta optimización es importante también porque **W** representa la solución encontrada, por lo que será la información que devolverán las hormigas del primer modelo de paralelización y la que tendrá que ser distribuida de nuevo al paso *takeOrdered*, por lo que tiene doble importancia que ocupe poco para aligerar la comunicación entre los distintos procesos de la tarea *Spark*.

- Los objetos a clusterizar muchas veces contienen información adicional no necesaria para nuestro algoritmo. Por ejemplo, si clusterizamos tweets en función de la localización geográfica, no nos interesa el contenido del tweet ni el autor. En un algoritmo de computación no distribuida, esto tiene menos importancia ya que la infomación no hay que desplazarla para usarla. En nuestro caso, es mejor aplicar un preprocesamiento que deje únicamente el identificador y la información necesaria para clusterizar para cada uno de los objetos antes de comenzar el algoritmo en sí.

- Las *Broadcast variables* que no se vayan a utilizar más deben eliminarse, puesto que ocupan un espacio que puede ser muy valioso cuando se manejan grandes cantidades de datos. En nuestro caso, cada vez que la matriz de feromonas se distribuye en una iteración, debe eliminarse de la caché la matriz de la iteración anterior para no ir acumulando información innecesaria, al igual que los centros de los clusters.

4

Fase experimental

En este capítulo analizaremos los resultados de la implementación. En primer lugar, en el apartado 4.1 se compararán las dos implementaciones realizadas, así como se describirán las pruebas de verificación que verifican el correcto funcionamiento de las implementaciones y sus resultados. Posteriormente, el apartado 4.2 comparará la implementación paralelizada con la implementación sin paralelizar y con la implementación de k-means que ofrece la librería *MLlib* de *Apache Spark*.

4.1. Pruebas de verificación

Una vez implementado el algoritmo ACOC y preparado para ser paralelizado en *Apache Spark* con los dos modelos planteados, se procede a comparar las dos implementaciones para comparar el rendimiento de ambas implementaciones. Para llevar a cabo esta tarea se han realizado un conjunto de experimentos con datos sintéticos, para analizar la convergencia de los diferentes modelos, así como la calidad de las soluciones encontradas.

Para el análisis de los dos algoritmos se han ejecutado diferentes problemas de tamaño variable. Inicialmente se ha partido de conjuntos de datos de un tamaño reducido, y se ha ido aumentando el tamaño del conjunto junto con otros parámetros para aumentar la complejidad de las pruebas y ver cómo se comportan cada una de las implementaciones. Dado que ambas implementaciones parten del mismo algoritmo sin paralelizar y dicha paralelización no modifica el algoritmo en sí, sino en que esclavo *Spark* se ejecuta cada parte del mismo, nos hemos centrado en los tiempos de ejecución para decidir qué implementación proporciona mejor rendimiento.

Los parámetros que se modificarán para comprobar la eficiencia de los algoritmos son los siguientes:

- Número de objetos m: El conjunto de datos sobre el que ejecutaremos el algoritmo es el parámetro que más afecta a la complejidad del problema. No solo hay multiples bucles en el algoritmo que deben recorrer los datos a clusterizar, sino que un conjunto de datos de mayor tamaño implica mayor carga de datos a trasladar entre los distintos nodos al ser paralelizado en *Spark*.
- Número de atributos n: Al igual que el parámetro anterior, la complejidad de los objetos a clusterizar afecta directamente a la complejidad del problema y al tiempo consumido

Parámetro	Valor
Número de atributos n	2
Número de hormigas elitistas r	20
Número de iteraciones I	10
Tasa de evaporación ρ	10 %
Influencia de la heurística β	50 %
Umbral de explotación q_0	80 %
Valor inicial máximo de las feromonas	0.5

Tabla 4.1: Parámetros no modificados durante las pruebas de comparación entre los dos modelos de paralización.

al ejecutarlo. En estas pruebas, con el objetivo de poder visualizar el resultado de la clusterización de forma gráfica, se ha mantenido en 2 atributos.

- Número de clusters g: En el apartado anterior hemos visto que la selección de cluster es la operación más costosa del algoritmo. Esta selección depende directamente del número de clusters en que queramos dividir el conjunto de datos inicial, por lo que cuanto mayor sea este número más complejo consideraremos el problema planteado.

- Número de hormigas R: En ambas implementaciones el número de hormigas afecta al número de subtareas *Spark* que se ejecutarán en cada iteración. Un mayor número de hormigas implica un mayor número de subtareas, con el consecuente incremento en los tiempos de procesamiento propios de *Spark* al tener que organizar esas subtareas.

- Número de iteraciones I: Al igual que el número de hormigas, el número de iteraciones afecta directamente al tiempo de ejecución. Al contrario que el número de hormigas, no hay mucha diferencia entre dos iteraciónes ejecutadas dentro de una misma ejecución del algoritmo, al menos a largo plazo. Teniendo esto en cuenta, el número de iteraciones se mantendrá fijo, dado que el valor que realmente nos interesa es el tiempo que tarda una iteración, no un conjunto de las mismas.

El resto de parámetros de configuración no afectan tanto al consumo de tiempo y recursos sino al error resultante de la clusterización, por lo que los mantendremos fijos para poder observar más claramente la influencia de los parámetros modificados. Los parámetros comunes a todas las ejecuciones se muestran en la tabla 4.1.

Los resultados de las pruebas realizadas se muestran en la tabla 4.1. En estos resultados se observa claramente como los tiempos de ejecución del modelo 2 son siempre muy superiores. Para interpretar estos resultados, hay que analizar el número de procesos y tareas *Spark* que cada una de las dos implementaciones necesita ejecutar. Vamos a tomar como ejemplo la primera configuración probada, que cuenta con 10 objetos, 2 clusters y 5 hormigas. La primera implementación lanza un proceso *Spark* por cada iteración, con una tarea por cada hormiga, por lo que una iteración con dicha configuración tiene 1 proceso y 5 tareas. En el caso de la segunda implementación, por cada hormiga y objeto se lanza un proceso, que contiene una tarea por cada cluster, es decir que en una iteración con dicha configuración se lanzan 10x5=50 procesos con 2 tareas cada uno. Conociendo estos datos, se puede deducir que el modelo 2 de paralelización lanza muchos más procesos *Spark*. El *overhead* que tiene cada uno de estos procesos al lanzarse reduce considerablemente el tiempo de ejecución del segundo modelo.

El *overhead* producido por lanzar los procesos *Spark* en la primera implementación depende del número de hormigas, mientras que en la segunda implementación depende tanto del número de hormigas como del número de objetos. Dado que el objetivo final del algoritmo ACOC es

# Objetos m	**# Clusters** g	**# Hormigas** R	**Modelo 1 Tiempo**	**Modelo 2 Tiempo**
10	2	5	0.58	13.09
		10	0.65	24.70
		15	0.66	37.49
	4	5	0.61	17.53
		10	0.58	31.75
		15	0.64	39.80
	8	5	0.59	13.39
		10	0.60	30.78
		15	0.64	44.48
20	2	5	0.55	29.31
		10	0.63	50.47
		15	0.66	78.32
	4	5	0.57	26.70
		10	0.61	48.73
		15	0.67	75.52
	8	5	0.58	25.36
		10	0.66	52.33
		15	0.75	75.72
40	2	5	0.59	50.06
		10	0.66	103.46
		15	0.75	153.14
	4	5	0.60	48.64
		10	0.70	108.86
		15	0.80	157.82
	8	5	0.63	50.43
		10	0.84	105.57
		15	0.93	150.80

Tabla 4.2: Resultados de las pruebas de comparación entre los dos modelos de paralización.

Parámetro	**Valor**
Número de objetos m	100
Número de atributos n	2
Número de clusters g	2
Número de hormigas R	[5, 100]
Número de hormigas elitistas r	[0, 20]
Número de iteraciones I	[5, 100]
Tasa de evaporación ρ	[10, 90] %
Influencia de la heurística β	[10, 90] %
Umbral de explotación q_0	[10, 90] %
Valor inicial máximo de las feromonas	[0.005, 0.5]

Tabla 4.3: Configuraciónes de ACOC probadas durante las pruebas de verificación.

poder aplicarlo a conjuntos de datos muy grandes, la segunda implementación pasaría demasiado tiempo preparando *Spark*, lo que reduce en gran medida el beneficio que puede aportar haber paralelizado el proceso. Debido a esto, en el resto de pruebas se ha desestimado el segundo modelo de implementación y se han continuado con el modelo 1 de implementación.

Entorno y configuración de las pruebas

Una vez seleccionado el primer modelo, se han llevado a cabo unas pruebas de verificación para comprobar que la implementación es correcta. Estas primeras pruebas se realizarán paralelizando sobre 8 nodos, usando una pequeña base de datos de 100 objetos de 2 atributos cada uno. Estos objetos se encuentra diferenciados claramente en dos grupos como muestra la figura 4.1.

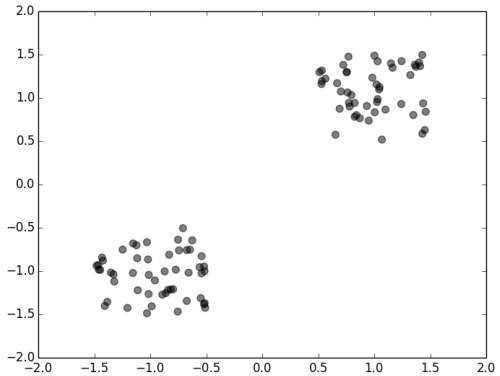


Figura 4.1: Distribución de la nube de puntos del problema de pruebas

Durante estas pruebas se han variado los parámetros de configuración de ACOC entre los rangos de valores que se muestran en la tabla 4.3, probando todas las combinaciones posibles con el objetivo de encontrar la configuración que proporciona el mejor resultado.

Resultados

Tras ejecutar la batería de pruebas con las combinaciones indicadas en la tabla 4.3, se han seleccionado los parámetros mostrados en la tabla 4.4 por proporcionar los mejores resultados.

Parámetro	Valor
Número de objetos m	100
Número de atributos n	2
Número de clusters g	2
Número de hormigas R	40
Número de hormigas elitistas r	20
Número de iteraciones I	100
Tasa de evaporación ρ	10 %
Influencia de la heurística β	50 %
Umbral de explotación q_0	80 %
Valor inicial máximo de las feromonas	0.5

Tabla 4.4: Configuración óptima de ACOC para las pruebas de verificación.

Hay que tener en cuenta que gran parte de la calidad de los resultados obtenidos con los algoritmos bio-inspirados como es ACOC dependen del problema concreto y de la elección de parámetros, por lo que en estas pruebas con datos sintéticos buscamos observar la evolución de la mejor solución para comprobar que el algoritmo funciona correctamente en función de su principal objetivo: maximizar la calidad de las soluciones encontradas (Eq. 4.1).

$$Q(X) = \frac{1}{\sum_{j=1}^{m} d(X_i, C_i)} \tag{4.1}$$

En primer lugar nos fijaremos en la evolución de la calidad de la mejor solución. Esta calidad se mide sumando las diferencias entre cada objeto y el centro de su cluster asignado e inviertiendo el resultado, aplicando la fórmula 4.1. Dado que es poco probable que todos los objetos se encuentren en el centro de su cluster, esta distancia no será cercana a 0. Esto provoca que los valores que observamos en la figura 4.2 sean tan bajos.

En la figura 4.2 vemos la evolución de la calidad de la mejor solución. Se puede observar como el algoritmo mejora el resultado obtenido a lo largo de las iteraciones, alcanzando la mejor solución antes de la iteración 40. A diferencia de los algoritmos genéticos, no contamos con una base de individuos que va mejorando, por lo que no podemos observar la evolución de la calidad media. Al mostrar únicamente la mejor solución se producen los grandes escalones que se observan en la gráfica.

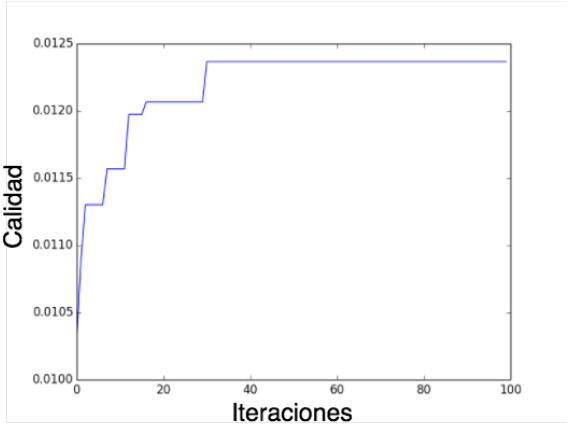


Figura 4.2: Evolución de la calidad de la mejor solución encontrada a lo largo de las iteraciones para el modelo 1.

Por otro lado, en la figura 4.3 se muestran los clusters resultantes, siendo el primer cluster

el que incluye los puntos de color azul y el segundo el que incluye los puntos de color rojo. Se puede observar como están claramente definidos y diferenciados.

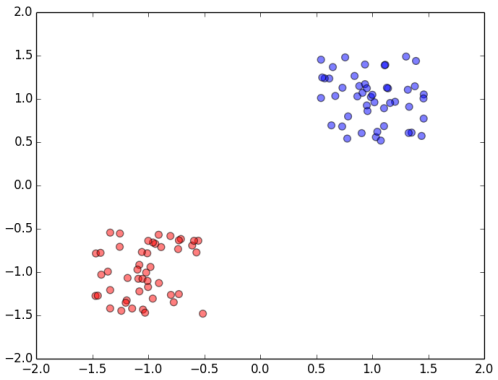


Figura 4.3: Resultado de la clusterización de las nubes de puntos originales con el modelo 1.

Estas pruebas han determinado que el modelo 1 del algoritmo funciona perfectamente al obtener una solución al problema de clusterización que se le plantea. En función del problema concreto a solucionar y del análisis que se haya realizado para establecer los parámetros la calidad de los resultados variará, pero como conclusión a este apartado cabe destacar que la implementación de ACOC distribuido en *Apache Spark* funciona perfectamente.

4.2. Comparación con otras implementaciones

Una vez seleccionada la implementación óptima de ACOC distribuido en *Apache Spark*, procederemos a comparar el algoritmo implementado con otros algoritmos para comprobar su eficacia. En primer lugar veremos la diferencia respecto al algoritmo ACOC sin paralelizar en el apartado 4.2.1. Después, en el apartado 4.2.2 veremos las diferencias con el algoritmo *K-means* que implementa *MLlib*.

4.2.1. ACOC no paralelizado

Ya hemos comprobado que la versión implementada de ACOC paralelizado en *Apache Spark* funciona correctamente. La versión de ACOC no paralelizado con la que compararemos sustituye la paralelización en *Apache Spark* por bucles tradicionales, por lo que el error resultante de clusterizar con ambas versiones es muy similar. Debido a esto, para comparar ACOC paralelizado en *Apache Spark* con ACOC sin paralelizar se analizarán los tiempos de ejecución de ambos. Para ello lanzaremos las distintas versiones con cantidades crecientes de datos a clusterizar para comparar los resultados. Las pruebas se realizan sobre un mismo ordenador, aprovechando 8 nucleos para paralelizar en el caso de la versión distribuida. La configuración en ambas pruebas es la obtenida durante las pruebas de verificación, que se puede observar en la tabla 4.5, donde el número de repeticiones representa el número de veces que se ha lanzado la prueba con cada algoritmo.

La gráfica 4.4 muestra el resultado de las 10 repeticiones de la prueba, mostrando los tiempos medios de las ejecuciones para cada cantidad de objetos a clusterizar. Se puede observar como los tiempos de ACOC sin paralelizar en verde aumentan exponencialmente, mientras que los resultados de ACOC paralelizado se mantienen mucho más estables. La segunda gráfica destaca

Parámetro	Valor
Número de repeticiones	10
Número de objetos m	[100, 2000]
Número de atributos n	2
Número de clusters g	2
Número de hormigas R	40
Número de hormigas elitistas r	20
Número de iteraciones I	100
Tasa de evaporación ρ	10 %
Influencia de la heurística β	50 %
Umbral de explotación q_0	80 %
Valor inicial máximo de las feromonas	0.5

Tabla 4.5: Configuración de ACOC en las pruebas de comparación con la versión no paralelizada.

la escalabilidad que cuenta la implementación en *Apache Spark*, ya que la diferencia entre los tiempos de ejecución aumenta casi exponencialmente.

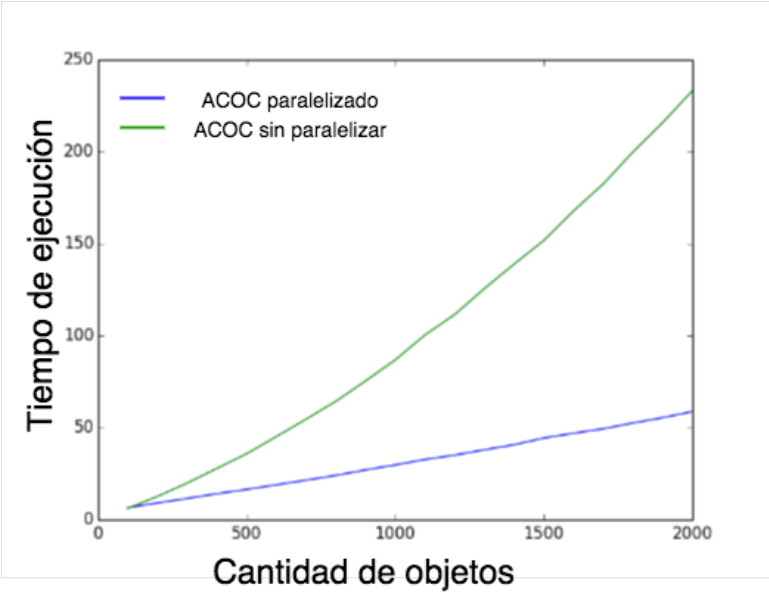


Figura 4.4: Evolución de los tiempos de ACOC paralelizado en azul y ACOC sin paralelizar en verde al aumentar la cantidad de objetos a cluserizar.

Como conclusión de este apartado, el algoritmo implementado logra superar con creces el algoritmo ACOC tradicional en tiempo de ejecución, obteniendo los mismos resultados.

4.2.2. K-means de MLlib

Para comparar *K-means* con nuestra implementación de ACOC paralelizado nos fijaremos en dos aspectos: tiempo y resultados. Las pruebas de tiempos las realizaremos sobre conjuntos de datos sintéticos. Las pruebas de calidad de las soluciones se han llevado a cabo utilizando datasets estandar obtenidos del repositorio de la Universidad de California, Irvine. Por último, un apartado de conclusiones resumirá la comparación con *K-means*.

Parámetro	Valor
Número de repeticiones	10
Número de objetos m	[100, 25000]
Número de atributos n	2
Número de clusters g	2
Número de hormigas R	40
Número de hormigas elitistas r	20
Número de iteraciones I	40
Número de repeticiones	10
Tasa de evaporación ρ	10 %
Influencia de la heurística β	50 %
Umbral de explotación q_0	80 %
Valor inicial máximo de las feromonas	0.5

Tabla 4.6: Configuración de ACOC al comparar tiempos con *K-means* de *MLlib*.

Parámetro	Valor
Número de objetos m	[100, 2000]
Número de atributos n	2
Número de clusters g	2
Número de máximo de iteraciones	40
Número de repeticiones	10
Tipo de inicialización	*Random*

Tabla 4.7: Configuración de *K-means* de *MLlib* al comparar tiempos con ACOC.

Comparación de tiempos

Las pruebas orientadas a medir los tiempos se han llevado a cabo utilizando los parámetros de las tablas 4.6 y 4.7. En el caso de ACOC, aplicaremos los mismos parámetros que los obtenidos durante las pruebas de verificación. Una vez más, el número de repeticiones representa el número de veces que se lanza el algoritmo con cada configuración, obteniendo la media de los resultados devueltos por dichas ejecuciones.

El resultado al lanzar el test con conjuntos de objetos de tamaño creciente entre 100 y 2000 objetos se puede observar en la tabla 4.8. Se puede ver claramente como los tiempos se diferencian desde el principio, ya que ACOC tarda mucho más en clusterizar los objetos. Los tiempos medios de ejecución de *K-means* en ningún caso superan los 0.3 segundos, mientras que los tiempos medios de ACOC comienzan en torno a los 10 segundos con 100 objetos y finalizan alcanzando más de minuto y medio con 2000 objetos a clusterizar.

Dado que ACOC debería destacar con conjuntos de datos mayores, se ha lanzado una prueba con conjuntos objetos de mayor tamaño, comenzando en 5000 objetos y alcanzando los 25000. Los resultados de esta prueba se recojen en la tabla 4.9. Una vez más, los tiempos de *K-means* son mucho más reducidos.

Comparación de la calidad de las soluciones encontradas

Para comparar la calidad de las soluciones encontradas se ha enfrentado a los dos algoritmos a problemas reales a solucionar. Los problemas seleccionados han sido la clusterización de los datos del Libras Movement Data Set[17] y del Gesture Phase Segmentation Data Set[18], ambos del repositorio de la Universidad de California, Irvine.

Número de objetos	**ACOC**	**K-means**
100	6.81	0.18
200	11.92	0.22
300	17.18	0.21
400	21.74	0.22
500	26.59	0.24
600	31.74	0.19
700	35.76	0.21
800	40.10	0.18
900	44.55	0.19
1000	51.34	0.19
1100	55.81	0.28
1200	59.32	0.21
1300	63.38	0.18
1400	67.62	0.19
1500	73.93	0.25
1600	80.11	0.19
1700	85.45	0.21
1800	88.28	0.22
1900	92.55	0.21
2000	97.75	0.19

Tabla 4.8: Tiempos medios de ejecución al ejecutar *K-means* y ACOC con conjuntos de datos generados aleatoriamente.

Número de objetos	**ACOC**	**K-means**
5000	196.48	0.31
10000	423.77	0.41
15000	659.85	0.42
20000	890.03	0.49
25000	1113.06	0.56

Tabla 4.9: Tiempos de ejecución en segundos en conjuntos de objetos de mayor tamaño.

Parámetro	**Valor**
Número de clusters g	[2, 40]
Número de hormigas R	[5, 100]
Número de hormigas elitistas r	[0, 20]
Número de iteraciones I	[20, 100]
Tasa de evaporación ρ	[10, 90] %
Influencia de la heurística β	[10, 90] %
Umbral de explotación q_0	[10, 90] %
Valor inicial máximo de las feromonas	[0.005, 0.5]

Tabla 4.10: Configuraciónes probadas de ACOC para los problemas Libras y Gestures.

Parámetro	**Valor**
Número de objetos m	360
Número de atributos n	90
Número de clusters g	15
Número de hormigas R	20
Número de hormigas elitistas r	5
Número de iteraciones I	50
Tasa de evaporación ρ	10 %
Influencia de la heurística β	20 %
Umbral de explotación q_0	90 %
Valor inicial máximo de las feromonas	0.05

Tabla 4.11: Configuración definitiva de ACOC para el problema Libras.

El primero de estos conjuntos de datos contiene 15 clases con 24 instancias cada una. Cada clase representa un tipo de gesto del lenguaje de signos oficial de Brasil. Cada instancia está representado con 90 atributos normalizados que contienen toda la información del movimiento. Este conjunto de datos UCI lo proporciona completamente procesado, en un único archivo cuyas filas representan instancias y cuyas columnas representan los valores de los atributos de las mismas, por lo que no ha sido necesario ningún preprocesamiento previo.

El segundo dataset es de mayor tamaño, conteniendo 9900 objetos con 32 atributos cada uno. Estos datos representan velocidades y aceleraciones de muñecas y manos de personas gesticulando al contar diversos relatos. Al igual que el conjunto anterior, no ha sido necesario ningún preprocesamiento.

La primera fase del experimento ha consistido en realizar numerosas pruebas con nuestra versión de ACOC para tratar de optimizar los parámetros a utilizar. Los rangos de los parámetros probados se muestran en la tabla 4.10. Una vez obtenidas las configuraciónes adecuadas de ACOC para cada problema, que se observan en las tablas 4.11 y 4.12, se ejecutaron ambos algoritmos para obtener las soluciones y comparar su calidad. En el caso de *K-means*, las configuraciones utilizadas se muestran en la tabla 4.13. La calidad de estas soluciones se obtiene calculando la suma de los errores cuadráticos de cada nodo con el centro del cluster asignado, con la fórmula 4.1 que hemos visto anteriormente.

La tabla 4.14 muestra la media y varianza de los resultados obtenidos tras lanzar cada problema 10 veces con cada algoritmo. Dada la fórmula aplicada para obtener los errores, es imposible obtener un error de 0. Las soluciones obtenidas por *K-means* son mucho mejores que las obtenidas por ACOC en este problema, aunque la diferencia en el conjunto Gestures no es tan grande. Hay que destacar que la calidad de las soluciones de ACOC se ve mínimamente afectada por la paralelización del mismo, ya que el algoritmo conserva su estructura y características.

Parámetro	**Valor**
Número de objetos m	9900
Número de atributos n	32
Número de clusters g	30
Número de hormigas R	20
Número de hormigas elitistas r	2
Número de iteraciones I	50
Tasa de evaporación ρ	15 %
Influencia de la heurística β	25 %
Umbral de explotación q_0	90 %
Valor inicial máximo de las feromonas	0.05

Tabla 4.12: Configuración definitiva de ACOC para el problema Gestures.

Parámetro	**Libras**	**Gestures**
Número de objetos m	360	9900
Número de atributos n	90	32
Número de clusters g	15	30
Número de máximo de iteraciones	10	10
Número de repeticiones	10	10
Tipo de inicialización	*Random*	*Random*

Tabla 4.13: Configuración de *K-means* de *MLlib* para los problemas de Libras y Gestures.

Como dato adicional, al contrario que cuando se manejaban datos sintéticos, en estos problemas ACOC converge más rápidamente, alcanzando la solución que consideraba óptima en las 20 primeras iteraciones del mismo.

Conclusiones

Como conclusión a este apartado, *K-means* ha probado ser más rápido y más eficaz en los problemas planteados. La implementación de *K-means* de *MLlib* está especialmente optimizada para su uso en *Apache Spark*, lo cual se aprecia en los tiempos increiblemente cortos en los que logra encontrar las soluciones. En las pruebas realizadas no ha relucido el componente estocástico de los algoritmos bio-inspirados como ACOC, que permite enfrentarse a problemas de dimensiones inmanejables para los algoritmos tradicionales. Una de las principales diferencias entre ACOC y *K-means* es que ACOC trabaja con los objetos de forma independiente, mientras que *K-means* los trata en conjunto como una matriz. Por tanto, la pregunta que plantean estos resultados es... ¿Que pasaría con los rendimientos si el problema planteado fuera tan grande

	ACOC	**K-means**
Media del error cuadrático en Libras	405.30	323.71
Varianza del error cuadrático en Libras	13.31	13.15
Tiempo medio en Libras	52.29	0.35
Media del error cuadrático en Gestures	205.61	184.55
Varianza del error cuadrático en Gestures	0.17	0.69
Tiempo medio en Gestures	1000.21	1.17

Tabla 4.14: Medias de las calidades de las soluciones obtenidas para el problema Libras por ACOC y *K-means*.

que no cabe en la memoria con la que *Apache Spark* optimiza sus tareas? Eso podría provocar una caida importante en el rendimiento de *K-means*, mientras que ACOC mantendría el suyo.

5

Conclusiones y trabajo futuro

5.1. Conclusiones

Este trabajo tenía una serie de objetivos mencionados en la introducción del mismo:

1. Estudio de los algoritmos de hormigas aplicados a las tareas de clustering.
2. Estudio de los diferentes aspectos del algoritmo ACO que puedan ser distribuibles.
3. Estudio de la plataforma Apache Spark, así como de los posibles algoritmos de ACO desarrollados en ella.
4. Desarrollo del algoritmo ACOC para tareas de clustering.
5. Validación del rendimiento del algoritmo diseñado.

A lo largo del desarrollo de este trabajo se han encontrado diversas limitaciones:

1. El desconocimiento inicial del funcionamiento tanto de Apache Hadoop como de Apache Spark.
2. El desconocimiento inicial de la optimización de procesos en el lenguaje de programación Python y el funcionamiento interno de este lenguaje.
3. La falta de una base de conocimiento centralizada de donde extraer el conocimiento sobre Spark necesario para llevar a cabo el desarrollo.

Durante el capítulo 3 se ha planteado el análisis e implementación de la paralelización de ACOC en *Spark*, obteniendo dos posibles implementaciones. La primera de estas implementaciones paraleliza las hormigas de cada iteración, mientras que la segunda implementación paraleliza el proceso de selección de cluster para cada objeto de cada hormiga.

Durante el primer apartado del capítulo 4.1 se han comparado las dos paralelizaciones implementadas, y se ha seleccionado la más eficaz de las dos, correspondiente al primer modelo de paralelización. En esta comparación se ha podido comprobar como la cantidad de procesos *Spark* que el segundo modelo necesitaba lanzar introducía demasiado *overhead* en la ejecución,

enlenteciendo considerablemente el algoritmo. En este mismo apartado se han lanzado pruebas de verificación con conjuntos de datos sintéticos. Estas pruebas de verificación han comprobado que el algoritmo paralelizado funciona correctamente, observando la evolución de la mejor solución a lo largo de las iteraciones. Los objetivos iniciales, por lo tanto, se han cumplido, dado que tenemos una versión de ACOC distribuida en *Apache Spark* que funciona perfectamente.

En el apartado 4.2 durante la segunda fase de las pruebas se ha comparado la versión distribuida de ACOC con la versión no distribuida. Hemos podido observar que la versión distribuida obtiene los mismos resultados que la versión tradicional, pero logra aumentar su escalabilidad enormemente, aumentando exponencialmente la diferencia entre las velocidades al aumentar la complejidad del problema. La versión distribuida de ACOC logra unos tiempos hasta 4 veces inferiores en las pruebas realizadas. No solo se ha conseguido implementar con éxito una versión distribuida de ACOC, sino que es claramente superior a la versión tradicional.

Por otro lado, se ha comparado la nueva implementación con la implementación de *K-means* de la librería *MLlib* tanto con datos sintéticos como con 2 conjuntos de datos reales. En todos los casos de prueba *K-means* ha obtenido resultados mucho mejores en un tiempo menor. Hay que tener en cuenta que la implementación de *K-means* probada es nativa de *Spark*, y ha sido desarrollada por sus mismos creadores, por lo que el conocimiento de las tecnologías usadas es mucho mayor y la eficiencia en el uso de las mismas no tiene comparación.

Está claro que la implementación distribuida de ACOC en *Apache Spark* obtenida tiene mucho por mejorar, pero se han sentado las bases para futuras implementaciones y mejoras.

5.2. Trabajo futuro

Como trabajo futuro se proponen los siguientes puntos:

- Continuar optimizando la implementación de distribuida de ACOC, tratando de minimizar los tiempos de procesamiento.
- Desarrollar distintas implementaciones de ACOC distribuido, probando la eficacia de distintos planteamientos y puntos de paralelización.
- Profundizar en el conocimiento de *Spark* y otras tecnologías similares, con el objetivo de hacer un mejor uso de las mismas.
- Llevar a cabo pruebas sobre bases de datos de un tamaño mucho mayor al probado. Para ello se planea desplegar un cluster de nodos sobre Amazon EC2, y comparar una vez más la eficacia de ACOC contra *K-means*. Al aumentar el tamaño del conjunto de datos, se busca alcanzar el punto en que las ventajas de ACOC en problemas complejos compensen la optimización de *K-means* en Spark.
- Realizar pruebas sobre problemas concretos, como clusterización de tweets, añadiendo un análisis semántico al proceso.

Bibliografía

[1] Vivien Marx. The big challenges of big data: as they grapple with increasingly large data sets, biologists and computer scientists uncork new bottlenecks. *Nature*, (PRESSCUT-H-2013-232):255–260, 2013.

[2] Michael Minelli, Michele Chambers, and Ambiga Dhiraj. *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses (Wiley CIO)*. Wiley Publishing, 1st edition, 2013.

[3] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[6] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization – artificial ants as a computational intelligence technique. *IEEE Comput. Intell. Mag*, 1:28–39, 2006.

[7] Yucheng Kao and Kevin Cheng. An aco-based clustering algorithm. In *Proceedings of the 5th International Conference on Ant Colony Optimization and Swarm Intelligence*, ANTS'06, pages 340–347, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[9] D. B. Fogel and L. J. Fogel. An introduction to evolutionary programming. In *Artificial Evolution*, volume 1063 of *Lecture Notes in Computer Science*, pages 21–33. Springer Berlin Heidelberg, 1996.

[10] James Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer, 2010.

[11] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61, 2014.

[12] Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Using river formation dynamics to design heuristic algorithms. In *Unconventional Computation*, pages 163–177. Springer, 2007.

[13] M. Dorigo and L. M. Gambardella. Ant colonies for the traveling salesman problem, 1997.

[14] A. Gonzalez-Pardo, F. Palero, and D. Camacho. An empirical study on collective intelligence algorithms for video games problem-solving. *Computing and Informatics*, In press, 2014.

[15] A. Gonzalez-Pardo, F. Palero, and D. Camacho. Micro and macro lemmings simulations based on ants colonies. In *EvoApp 2014*, volume In press, 2014.

[16] M. Dorigo. *Optimization, Learning and Natural Algorithms (in Italian)*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992.

[17] Uci - libras movement data set. `https://archive.ics.uci.edu/ml/datasets/Libras+Movement`. Accessed: 2015-08-12.

[18] Uci - gesture phase segmentation data set. `https://archive.ics.uci.edu/ml/datasets/Gesture+Phase+Segmentation`. Accessed: 2015-08-12.

[19] Uci - bag of words data set. `https://archive.ics.uci.edu/ml/datasets/Bag+of+Words`. Accessed: 2015-09-22.